



TM

freeBSD

TM

March/April 2017

**JOURNAL**

**Configuration  
Management**

the

# Pipe Dream

- CFEngine
- Puppet
- Vagrant
- SaltStack

**ALSO**

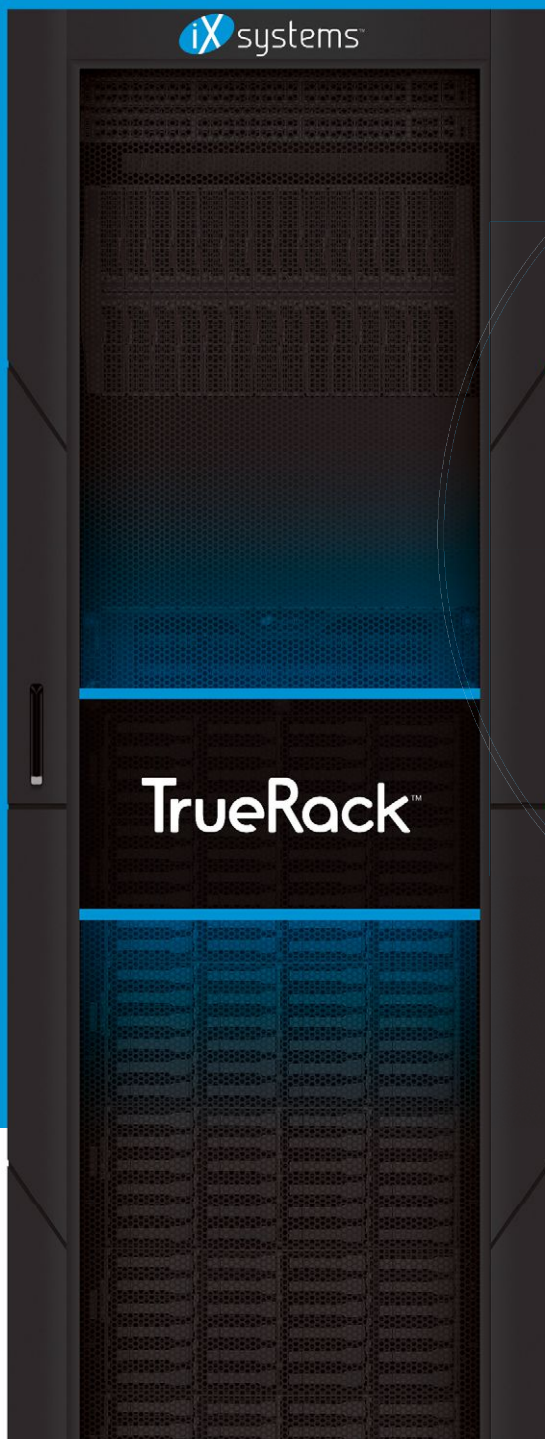
**DTRACE**

A Book Review



# Simplify your Data Center

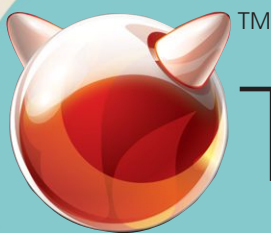
**Meet TrueRack™** — A powerfully flexible rack-scale architecture that takes the guesswork out of building large scale data center applications.



- Converged Infrastructure
- Customizable for Virtualization, Big Data, Cloud & Hyperscale
- Up to 70% Lower TCO Than AWS
- Scalable and Repeatable Deployments

For more information on TrueRack, visit **[iXsystems.com/TrueRack](https://iXsystems.com/TrueRack)** today





# Table of Contents

## Foundation Letter

**3** Configuration management—start here!  
*By George Neville-Neil*

## svn Update

**38** Developers started work to add support for creating reproducible builds back in 2013, and in the past couple months, we've seen several commits that bring this work even closer to completion. *By Steven Kreuzer*

## New Faces

**40** In this installment, the spotlight is on Johannes Dieterich, who became a ports committer in January, and Mahdi Mokhtari, who became a ports committer in February. *By Dru Lavigne*

## Conference Report

FOSDEM 2017 FOSDEM (Free and

**42** Open Source Software Developers' European Meeting) is a noncommercial, volunteer-organized European event, centered on free and open-source software development. *By Benedict Reuschling*

## Book Review

**46** *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD* by Brendan Gregg and Jim Mauro.  
*By Joseph Kong*

## Events Calendar

**48** Through June 2017  
*By Dru Lavigne*

## Configuration Management

### The Pipe Dream

Once you touch the configuration management pipe dream, you'll hang on to it for the rest of your career. *By Michael W Lucas*

### Yes, CFEngine Can Manage That

Next to shell scripts and post-it notes on the monitor (and maybe some obscure offering from IBM), CFEngine is the oldest and most mature of all configuration management solutions out there. *By Phillip Jaenke*

### Puppet on FreeBSD

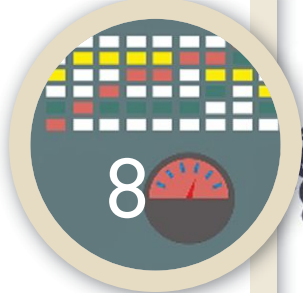
One of the most popular configuration management tools out there, Puppet started something of a revival in the configuration management space. *By Brad Davis and Andrew Fengler*

### Vagrant

Vagrant is a command-line utility for building and managing portable development environments, and is often used for setting up development and test environments. *By Steve Wills*

### SaltStack on FreeBSD: A Primer

SaltStack is a Python-based configuration management solution that is feature rich and has a reputation for ease of use and scalability. *By Peter Wright*



# Support FreeBSD®



## Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.  
[freebsd.foundation.org/donate](https://freebsd.foundation.org/donate)



- John Baldwin • Member of the FreeBSD Core Team and Co-Chair of *FreeBSD Journal* Editorial Board
- Brooks Davis • Senior Software Engineer at SRI International, Visiting Industrial Fellow at University of Cambridge, and past member of the FreeBSD Core Team
- Bryan Drewery • Senior Software Engineer at EMC Isilon, member of FreeBSD Portmgr Team, and FreeBSD Committer
- Justin Gibbs • Founder and President of the FreeBSD Foundation and a Senior Software Architect at Spectra Logic Corporation
- Daichi Goto • Director at BSD Consulting Inc. (Tokyo)
- Joseph Kong • Senior Software Engineer at EMC and author of *FreeBSD Device Drivers*
- Steven Kreuzer • Member of the FreeBSD Ports Team
- Dru Lavigne • Director of the FreeBSD Foundation, Chair of the BSD Certification Group, and author of *BSD Hacks*
- Michael W Lucas • Author of *Absolute FreeBSD*
- Ed Maste • Director of Project Development, FreeBSD Foundation
- Kirk McKusick • Director of the FreeBSD Foundation and lead author of *The Design and Implementation* book series
- George V. Neville-Neil • Director of the FreeBSD Foundation, Chair of *FreeBSD Journal* Editorial Board, and coauthor of *The Design and Implementation of the FreeBSD Operating System*
- Hiroki Sato • Director of the FreeBSD Foundation, Chair of Asia BSDCon, member of the FreeBSD Core Team, and Assistant Professor at Tokyo Institute of Technology
- Benedict Reuschling • Vice President of the FreeBSD Foundation and a FreeBSD Documentation Committer
- Robert Watson • Director of the FreeBSD Foundation, Founder of the TrustedBSD Project, and University Senior Lecturer at the University of Cambridge

**S&W PUBLISHING LLC**  
PO BOX 408, BELFAST, MAINE 04915

- Publisher** • Walter Andrzejewski  
walter@freebsdjournal.com
- Editor-at-Large** • James Maurer  
jmaurer@freebsdjournal.com
- Copy Editor** • Annaliese Jakimides
- Art Director** • Dianne M. Kischitz  
dianne@freebsdjournal.com
- Office Administrator** • Michael Davis  
davis@freebsdjournal.com
- Advertising Sales** • Walter Andrzejewski  
walter@freebsdjournal.com  
Call 888/290-9469

*FreeBSD Journal* (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).  
Published by the FreeBSD Foundation,  
5757 Central Ave., Suite 201, Boulder, CO 80301  
ph: 720/207-5142 • fax: 720/222-2350  
email: info@freebsdjournal.org  
Copyright © 2017 by FreeBSD Foundation. All rights reserved.

This magazine may not be reproduced in whole or in part without written permission from the publisher.

## CONFIGURATION MANAGEMENT— *Start here!*

This issue's articles focus on the most popular configuration management systems, all of which are well supported on FreeBSD. Leading the pack, Michael Lucas, author of many well-known books on the BSDs, contributes his thoughts on the topic of configuration management in general. If you're planning your first foray into configuration management, I suggest you begin with his intro piece before moving on to the system-specific articles on CFEngine, Puppet, SaltStack, and Vagrant. Each system has pluses and minuses, but all have one thing in common—they can help you deploy and manage FreeBSD.

Over the past few years, a hot topic in the open-source world has been reproducible builds. Originally brought to light by security issues such as heartbleed, open-source projects take this topic very seriously. The FreeBSD Project has been heavily involved in the process, with Project and Foundation representatives attending and contributing at various reproducible build conferences and meetings. That work is now paying off in commits going into FreeBSD, which Steven Kreuzer covers in his latest svn Update column.

Joseph Kong reviews Brendan Gregg and Jim Mauro's *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. I know this book well, as I have been using it in the courses developed for <http://www.teachbsd.org>. I won't spoil the ending though—go read and enjoy the review.

The conference season is upon us, and the first conference report for this year is by Benedict Reuschling, who volunteered at FOSDEM, the largest open-source conference in Europe. Covering a conference with 5,000 attendees and multiple interesting tracks over a two-day period is no small feat. But Benedict makes us feel we're right there in the thick of the open-source soup. And by the way, there is still plenty of time to register for BSDCan, scheduled to take place in Ottawa, Canada, in June. We hope you'll join us there. Stop by our booth and say, "Hi." Many from the *FreeBSD Journal* team will be in attendance, and we'd love to chat with you about the *Journal* or whatever else is on your mind.

George Neville-Neil

**Editor in Chief of the *FreeBSD Journal***

Director of the FreeBSD Foundation



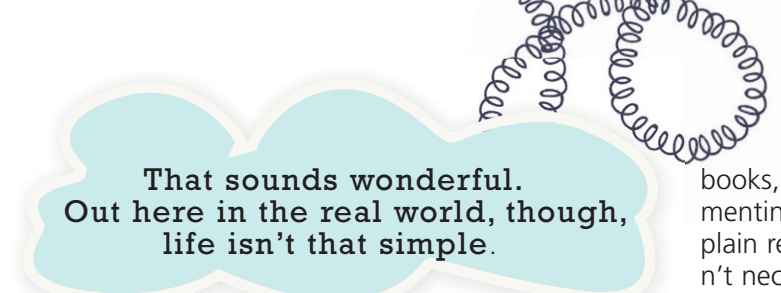
The Configuration  
Management

# Pipe Dream

by

**Michael  
W. Lucas**

Configuration management is a dream. A glorious, beautiful dream. Define all of your server characteristics in a central database! Deploy cloud servers as workload They'll all be identical! Behave identically! Everything will just work! Absolute reproducibility, absolute predictability!



**That sounds wonderful.  
Out here in the real world, though,  
life isn't that simple.**

Most of my career has been spent in large enterprises: sometimes as an employee, but most often as a hired-gun consultant. These are companies where the server controlling the door lock runs Novell NetWare, because they really knew how to build hard drives in 1993. These organizations deploy applications to solve specific problems. Most often, they buy applications from vendors with promises that the programs require only minimal customization. Some applications have teams of sysadmins dedicated to their care and feeding, while other applications were benignly neglected until they crashed and died and the whole company noticed.

One thing that all these applications had in common was that the server was set up specifically for that application. The salesperson would tell management that "Our *Cures Everything* ERP software needs a Sun Megajillion with 40 gadzooks of REM, and for your workload you'll want a crystal meth coprocessor." We techs would do our best to translate sales speak into actual hardware and software requirements, and then configure the equipment as per the instructions.

To be fair, the specs weren't always that ambiguous. Some were disturbingly specific, requiring hardware that had gone out of manufacture a couple years ago.

And vendors were always sure to mention that if our company's techs couldn't understand their perfectly clear requirements, for reasons of overwork or simple mental deficiency, their \$500/hour consultants could come in and deliver us a completely configured host. Sure, it would be managed via telnet, and lack all hard drive redundancy, but the application itself would work just dandy. Plus, the read-write SNMP string of "public" meant anyone in the company could tune the server, so even we lazy oafs couldn't fail to successfully support it.

So not every application was a special snowflake. Some of them were special hurricanes. Or special dingoes with very special rabies.

"Artisanal systems administration" might sound like a joke, but it sure feels like hard cold reality. The idea of configuration management in an environment like this is laughable. It goes into the same category as "if we were starting over and building from scratch, things would be perfect."

No, we had to maintain unique configurations for every host.

All these organizations had standards, mind you. The security and sysadmin teams maintained run-

books, or policy docs, or standards docs, documenting how every service should be configured, in plain readable English. Plain readable English doesn't necessarily translate to `/etc/rc.conf` settings. And never mind that Red Hat didn't support `pam_mkhomedir`, or that the critical application that relies on PHP version 2 and MySQL 3 dot mumble on this repurposed desktop running FreeBSD 4.4 absolutely requires SSH version 1.<sup>1</sup>

Sysadmins follow standards as closely as possible. We tune each box as best we can.

And hope. There's a whole great big scoop of hope, every day.

My last job was, in some way, the pathological end case of this management philosophy. A small, friendly company seemed like a great change. The people were fantastic—I'd known them personally for years, and some of them for decades. Everyone knew their area of responsibility, and yet was willing to help others out. The boss was highly technical and understood exactly how the simplest things could go horribly wrong. If I had to make an emergency visit to the data center at bite-me-o'clock, nobody expected to hear from me for a day or two. The company president considered keeping the fridge stocked with caffeine one of their most vital tasks. Dress code? "You must be dressed."

And absolutely no technical standards.

"I need service X? Fine, I'll stand up a VM for it. This VM server looks like it has capacity. I heard Mint is the new hotness; let's see what all the noise is about. It works, great; let's live with it for eight years!"

"Oh, hey, here's an Itanium box! Let's plug it in and see what we can run on it! Will Asterisk work? Cool, this customer is having trouble, let's route their calls through it and see if it helps."

"The whole company uses Apache, but I hear good things about nginx..."

They'd had a coherent design, some 15 years ago. But years of growth and expansion and purchases had buried that design under layers of infrastructure. And nobody had taken the time to automate anything.

The sign over the entrance to IT Hell reads "Every decision seemed sensible at the time."

And in walks "yours truly," the new Head Honcho of Systems Administration. I got a convenient list of all the servers in the company. All the servers anyone could think of at the moment, that is. For a couple weeks, that list grew daily. For months afterwards, that list grew weekly. One server appeared more than a year after I started.

I found myself walking datacenters with a crash cart, plugging the console into one host after another to figure out what it was. Sadly, I lost a bet by never discovering anything running VMS or Ultrix.



One man can't realistically run all of this. It just can't be done. And everyone had their own preferences for everything from operating systems software versions. The company's stated policy was to indulge those preferences.

Weirdly, company morale was good. Frustrations were high, but morale was good.

I grabbed my packet sniffer and started straightening this tangle.

I started with the obvious, the 15 or so unmaintained name servers on different parts of the network. I installed three new authoritative and three recursive nameservers, properly placed around the network. I updated `resolv.conf` on every server. Every server I knew about, that is.

I found the other servers when I shut off the unmaintained nameservers.

So: add them to the list of servers. Deal with the customer fallout. Apologize to the engineers whose day I've ruined. The staff understood what I was doing, mind you. They knew the situation needed improving... but that didn't mean they had to like the inevitable pain.

I deployed centralized authentication. One server at a time. Across multiple operating systems. You haven't lived until you've installed nine different LDAP authentication modules on 87 different operating systems, all by hand, one host at a time.

Something clearly had to give.

But centralized authentication opened up new possibilities.

And this is where we get to configuration management.

Configuration management isn't just for whole enterprises. Configuration management can start small, and seemingly unambitiously.

If your organization truly doesn't have configuration management, a one-man army can get it started. One configuration management advocate can implement a solution for their own chunks. Patience and demonstrated success will do the rest of the work for you.

I stood up one more host, running Ansible. (I chose the software that made the most sense to me. Pick whatever's best suited to your environment and your requirements.) I didn't start with deploying whole machines via the cloud. Instead, I centralized the contents of `/etc/resolv.conf` on all of the infrastructure machines I'd just stood up. This let me work out the countless small problems you always find when trying something new. I then brought `sshd_config` under configuration management.

Suddenly, I could deploy a change to my new infrastructure servers by editing one file and running one command. SSH isn't one of those services you have to change often, but changing

dozens of servers manually can ruin entire days.

Right about then, we found an intruder had broken into some of the servers. The intruder didn't appear to be interested in inflicting any damage on our systems, merely running some IRC bots.

Still, the intruder had to go.

From what I could determine, the intruder had broken in via SSH. Yes, some of the hosts still used password authentication. And some of the passwords were, shall we say... poorly chosen.

And it turned out that this wasn't the first time it had happened.

The staff finally decided that SSH passwords were unacceptable and we needed to allow only key-based authentication. I congratulated them on joining the 1990s, ran an Ansible command, and reported that my new servers were all updated but that I'd need more time to do all the others.

That started the configuration management discussion.

Each managed `sshd_config` file looked weird. They didn't have all the usual commented-out configuration statements, but instead looked much like this.

```
#Configuration Managed by Ansible
#Manual Changes Will be Overwritten
#You Was Warned
Port 22422
PasswordAuthentication no
Subsystem      sftp    /usr/libexec/sftp-server
```

The configuration file didn't need the commented-out defaults, because nobody would be editing this configuration file on this host to change anything. Any edits needed to happen on the configuration management host.

The discussion quickly turned to "how do we do this everywhere?"

And here's the trickiest part of deploying configuration management.

You never say, "Let me go do it."

You say, "I can do this on every host that authenticates via LDAP, but I need everyone to agree that they will no longer configure the SSH server. Once I bring a host's SSH under config management, it stays there. Everything gets documented in the configuration management system. It's no longer your problem, but you don't play with it anymore."

"But what if things go horribly wrong late sometime and you're not around?"

"Then you fix them and talk to me in the morning. But the next time I deploy a config update, your changes will get wiped out and I ain't listening to you whinge about it."

It wasn't quite as easy as that. Each



sshd\_config had to include or exclude statements based on the operating system, to cope with each OS's peculiarities. This little project exposed a few hosts that weren't connected to LDAP—not because I didn't know about them, but because the system couldn't handle the downtime to make that switch.

Weirdly, when you can demonstrate that a change will reduce an engineer's workload, they become really cooperative. Hosts that couldn't withstand any downtime at all to make the LDAP switch suddenly became available.

At 3 a.m. on a Sunday, yes, but still.

We weren't deploying virtual machines automatically. But moving just one service to configuration management changed everything.

"Can you push my authorized\_keys to all my hosts?"

"Of course I can. You know, nobody wants to upload a new authorized\_keys everywhere. What if I take on all the authorized\_keys maintenance? And we all know that letting users edit their own key files can be a security risk. We could move them to /etc/ssh/keys, lock these hosts down a little more." People became more likely to generate client keys more often than once a decade.

Failure of a host's name service led to bringing everything's resolv.conf into configuration management.

An LDAP change? Get the client LDAP set up in config management.

I had expected a slow increase in people asking if we could bring services under configuration management. Turns out I had underestimated the intelligence of my coworkers. One day, configuration management was a pipe dream. The next day, dang near everyone asked me to run those pipes through *everything*.

Retrofitting complete configuration management onto an existing artisan system isn't easy. It's actually pretty terrible.

It's much easier to spin up new virtual machines and deploy to those.

Most of those random hosts needed replacing anyway. It's just that configuring the services was a pain so nobody wanted to touch them.

Best of all, we knew certain things were no longer a problem. My boss's new favorite phrase became "we are *done* with this problem." (True, he also used it for things like the scheduled cola delivery, but that's not the point.)

Morale stayed good. Frustration levels remained the same, but folks were frustrated about different things. The lesson here is that some people aren't happy unless they're frustrated.

And we all lived happily ever after.

Except for the part where we deployed a whole bunch more VMs, because automation made it possible to manage them all easily. But there were pizza

and paychecks, so, yeah, still pretty happy.

The weird thing is, I'm not the only sysadmin who's had this story.

Everyone who starts with configuration management, even at a small scale, discovers that it's an incredibly powerful tool. You don't need to start with a new, empty environment. Start with your own pain points. Maybe you have a whole bunch of Apache servers and you need to make sure that everyone has an up-to-date TLS includes file. Maybe you want to be sure that every server you manage has a certain list of packages installed, or that the sudo package is always current.

Our brains all know that automation makes everything easier. But running that first command and seeing your updates flow across your network tattoos that knowledge onto your bones.

Configuration management is not perfect. A configuration management system can deploy mistakes at the speed of Ethernet. (Do an Internet search for "sudo bake me a cake" to find Jan-Piet Mens's amusing failure to manage sudo. It's amusing, because it's not me.)

Using configuration management makes testing far more important. These days, when even cheap laptops can run a fleet of virtual machines, testing is a matter of time and attention—always scarce, yes, but better than requiring time, attention, and thousands of dollars of hardware.

Best of all, managers notice that configuration management helps. Configuration management reduces outages from human inconsistency. It helps expose problems. Eventually, you'll hear your boss tell a salesperson, "That sounds nice, but can we manage it with Ansible/Puppet/Chef/whatever?" You can then concentrate on explaining that "40 gadzooks of REM" is not a real thing, and that any salesperson who says you need it is incompetent.

This issue includes articles on using Vagrant, Puppet, SaltStack, and CFEngine. They all work well with FreeBSD, either as a server or a client. These tools are the very definition of a force multiplier, letting one sysadmin support many more servers than without. Yes, the lucky sysadmin who gets to start out with a blank slate might support hundreds of on-demand cloud servers, but the rest of us can make our most tedious tasks evaporate.

Once you touch the configuration management pipe dream, you'll hang on to it for the rest of your career.

**MICHAEL W LUCAS** is the author of several books on FreeBSD, including *Absolute FreeBSD* and the *FreeBSD Mastery* series. Learn more at [www.michaelwlucas.com](http://www.michaelwlucas.com).

<sup>1</sup> I truly wish I was joking.

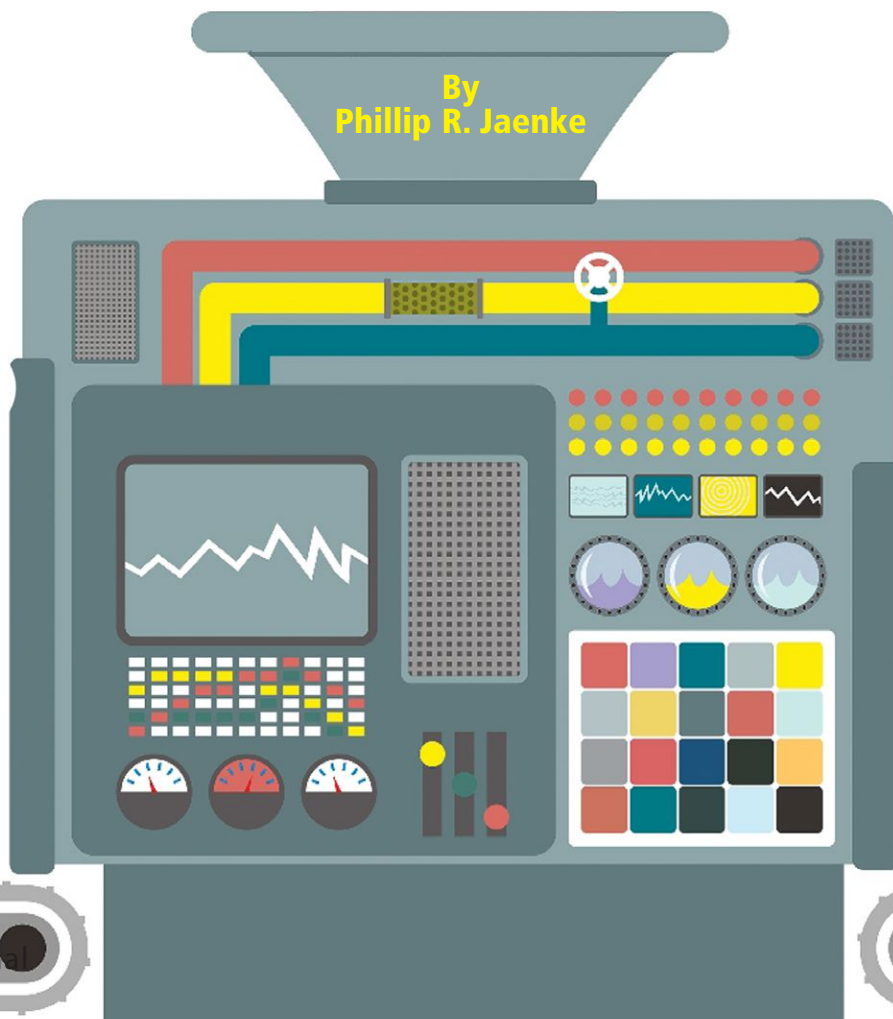
# Yes, CFEngine

## CAN MANAGE THAT

Configuration management is, as you might have guessed, currently a hot topic in IT.

With the advent of people spinning up VMware guests by the dozen and another 50 instances in AWS for good measure, it's become more of a necessity than ever before. Once a setup with 200 environments on 2 dozen hosts (because jails really are that great) was considered a large environment. It was also something a pair of skilled system administrators could easily maintain, because they built every one of those environments. Sure, you had lots of one-offs, but you could easily track them with a text file here and a post-it note there. (Do not disable password logins on clx90a because they access that from the AS/400.)

By  
Phillip R. Jaenke







These days, “large” environments start at about 20,000 hosts and only go up from there. As you might imagine, no amount of skill will ever make up for that kind of volume. And these environments often go back more than a decade. It is guaranteed that at some point you will poke your head into a troublesome host only to find out that it’s a Sun E420R running Solaris 2.5.1, despite the company having standardized on FreeBSD and Windows more than a decade back. Also, it’s the one singular system storing your entire login database—for all the employees and customers.

These problems are certainly not new or unknown. The scope and scale have changed tremendously over the years, but the notion that setups like this exist and cause untold numbers of WTF-o’clock calls is almost as old as a moth in the relay. A gentleman doing a post-doctoral fellowship at Oslo University College realized that this was a real problem, so he became a monk instead—wait, no, that’s just what a lot of us wished we could do. Instead, Mark Burgess decided to come up with a solution, and, in 1993, released the very first iteration of CFEngine.

Next to shell scripts and post-it notes on the monitor (and maybe some obscure offering from IBM), CFEngine is the oldest and most mature of all configuration management solutions out there. And not by a little, either. The next oldest you’ll find discussed in this issue is Puppet, the initial release of which occurred 12 years *after* CFEngine. And immediately, those of us who have been down the single glass of pain and “mature solution” road cringe at the horror of 1990’s legacy C.

While it’s true that CFEngine is written in C, it’s for portability and performance reasons. Other configuration managers are primarily or exclusively based around their own code and the general concept of configuration management. CFEngine’s longevity has come about because it is based around two key concepts: convergence and Promise Theory. Rather than say “things are bad, make them all the same in an easy fashion,” Mark Burgess sought to understand the ad hoc choices that had been made initially and a method for understanding those choices.

You can read about all that on Wikipedia of course. But that is the nuts and bolts of why CFEngine has given users a consistent and sta-

ble set of expectations and behavioral concepts for more than two decades and two complete rewrites. Because it is an implementation of concept and theory, the details of how it does so are far less important.

## A Useful Hello World in CFEngine

Because CFEngine uses bidirectional communications (agent and server model) and is specifically targeted at heterogeneous environments, our example here will show you how to use CFEngine’s Domain Specific Language to write a ‘Hello, World’ promise that tells you a bit about the system as well.

To start, you need to understand how the process works. First, you use your policy files to define the desired state. The agent then ensures the state (every 5 minutes). Then the server may optionally verify the desired state, if you’re using CFEngine Enterprise. ObPlug: Enterprise is completely free for up to 25 hosts, but unfortunately requires your policy hub be Linux.

## The Promise

```
## This is a CFEngine Hello World Promise
bundle agent hello_world
{
    reports:
        any::
            "Hello World";
}
```

This promise establishes a bundle called ‘hello\_world’ that will report on any class of hosts. Now we’re going to do something a bit more elaborate leveraging CFEngine’s built-in classes, called hard classes: (next page)

```
## Fancy CFEngine Hello World Promise
bundle agent hello_world_freebsd
{
    reports:
        any::
            "Hello World";
        freebsd::
            "Thanks for reading FreeBSD Journal!";
        windows::
            "I think I got lost.";
}
```

This policy will do two additional steps. If the system is a member of the FreeBSD hard class, it offers a different greeting from a windows member, while all others offer the default greeting. Let's see this policy in action on two different hosts now.

```
root@freebsd11 # /var/cfengine/bin/cf-agent --no-lock --file
./hello_world.cf --bundlesequence hello_world_freebsd
R: Hello World
R: Thanks for reading FreeBSD Journal!

[root@centos7] # /var/cfengine/bin/cf-agent --no-lock --file
./hello_world.cf --bundlesequence hello_world_freebsd
R: Hello World

root@freebsd11# /usr/local/sbin/cf-agent --no-lock -D windows --file
./hello_world.cf --bundlesequence hello_world_freebsd
R: Hello World
R: Thanks for reading FreeBSD Journal!
R: I think I got lost.
```

As you can see, CFEngine's class mechanism is actually quite easy to understand and leverage. This enables you, as the person in charge of preventing chaos, an easy and reliable means to differentiate your hosts within your policies. Because you can also use promises to define soft classes by virtually any means you like, this gives you a combination of flexibility and consistency that many other configuration management solutions often lack or make difficult to implement.

But as with most Hello World examples, this barely scratches the surface of the capabilities available.

## Promises and Policies and Bundles, Oh Why?!

When you work with CFEngine, you might be editing files, but those files are inextricably linked to the core concept of Promise Theory. The gist of Promise Theory is that autonomous actors will achieve voluntary cooperation, which sounds

about as far from actually doing configuration management as you can possibly get. Words like "voluntary" and "autonomous" and "cooperation" sound weird in this context.

So, to start, let's break down what Promise Theory really is. As you might have guessed, Mark Burgess proposed it as a method of solving problems in obligation-based schemes for policy-based management (or more accurately, to explain CFEngine's operational model). Rather than have configuration directives dictated from a single server or single group of authoritative servers, every member in the group has autonomy of control. That is, they can't be forced into specific behaviors by deterministic commands. Instead, the agent can only make promises about its own behavior and no other system's behavior.

CFEngine agents are not slaving themselves to a single central server, but rather, an agent promises to retrieve data and act on instructions from another server. A good analogy would be a typical everyday occurrence on the job. Your intern promises he will fix the problem on all 50 systems if you tell him how to do it. You give the intern a list of basic steps—edit

these files, restart these daemons. The intern then performs the steps exactly as you wrote them. Because he is an intern, you then verify that the results are what was desired. That is an (admittedly very simplified) example of Promise Theory in action.

You're not cooperating due to any explicit obligation or requirement aside from wanting the system in a specific state; an intern does not make any promises about any actions besides his own; you do not sit there watching over the intern's shoulder or enter the commands for him—you simply tell the intern what you want him to do; the intern reports back only the specific information he directly knows; you then validate the instructions had the desired result. I know the whole "intern volunteering and actually understanding what he did" part is probably farfetched, but you get the point.

Now imagine an army of helpful, cooperative, knowledgeable interns. That's pretty much the CFEngine model. Thousands of interns asking



root, “Hey, what do you want me to do? Okay, did it, here’s the result.”

Having an understanding of that, it becomes much easier to understand how promise, policy, and bundles fit together in CFEngine. In the simplest terms, a policy is a collection of promises. If I have promises called “abc” and “123”, I can then combine them into a policy called “321cba” that incorporates both of those promises. If you apply a policy with no promises, nothing will happen.

Our Hello World above is an example of a promise. To incorporate that promise into a policy, I would use a body statement with a bundlesequence like this:

```
body common control
{
    # These are the promises we want the agent to act on
    bundlesequence => { "hello_world_freebsd" };
}

bundle agent hello_world_freebsd
{
    ...
}
```

A bundlesequence can be thought of as a policy. CFEngine will execute the promises contained within the bundlesequence statement in order. I can then use those promises to control the behavior of other promises. For example, I might use a promise that finds out if the system’s hostname begins with the letters qa, and then installs a different sudoers file if that promise is true. I can also place the hostname check entirely within the sudoers promise itself.

But what is a bundle, and how does it fit into things? Especially since we just said promises go into policies! Well, a bundle is a means of collecting similar or related promises into a single promise. For example, my policy may promise that any::systems receive the bundle “standard\_users”. Within that bundle, I would then define appropriate promises to cover each user and OS.

Bundles, in other words, provide a means to make many promises based off a single policy decision. The caveat of a bundle is that each promise within the bundle cannot be treated as a separate promise, which is to say that you cannot reuse any of those promises outside of the bundle, unless you make it a separate stand-alone promise. You also cannot use it if all the promises need to be checked before another bundle.

This makes bundles very useful for broad stroke configuration items. But a bundle promise can do everything a stand-alone promise can do as well. So, those broad strokes can actually be extremely

system specific—such as setting the correct IP address, registering it in Active Directory, or even joining it to a cluster. It all depends on how you write the promises themselves.

## So How Do I Stand It Up?

As we just mentioned (and you probably noticed), CFEngine is an agent-based architecture. Agents will check in with a server—this server is called your Policy Server. The Policy Server is the central repository for not only all your policy files, but any other file you want to distribute. Additionally, with Enterprise, you can use it to distribute the

CFEngine binaries or packages to all your agent systems—automatically selected by OS, version, and endianness thanks to hard classes!

Except for Enterprise (called Nova Hub), which has additional interfaces and reporting tools, literally any system that can run CFEngine can be a Policy Server. There are only two

basic requirements. One, it must run a version of CFEngine that is compatible with your clients. Two, it must be able to hold any files you want to distribute using CFEngine itself. (Obviously if you prefer to use HTTP/HTTPS distribution or NFS for large files, you can cheerfully ignore this.)

Setting up your Policy Server is similarly simple in the grand scheme of things. First, you will need to build and install CFEngine from ports. Then, build and install the corresponding CFEngine masterfiles from ports. So, for 3.7 you would use **sysutils/cfengine37** and **sysutils/cfengine-masterfiles37**.

Note that because of some quirks, CFEngine cannot actually run from /usr/local so you will need to first copy or symlink the binaries from

**/usr/local/sbin/ to /var/cfengine/bin/.**

Finally, enable CFEngine in your rc.conf and run the command **/var/cfengine/bin/cf-agent --bootstrap --policy-server 127.0.0.1**

You have now successfully built your Policy Server!

Really, that’s it. You’re now ready to start writing your policies and connecting clients—which is as simple as changing the IP address of the policy-server argument you just ran above. CFEngine’s communications are fully encrypted and automatically handle all the hairy work of creating and maintaining secure keys for you.

One of the things that often catches people off guard about CFEngine is that clients are survivable.

Let's say you need to take your Policy Server down to replace a failed DIMM. (And not because Jr. SysAdmin Jimmy ran a random shell script he downloaded off the Internet as root. By the way, there is now an opening for a Jr. SysAdmin.) But your environment keeps trucking and keeps enforcing.

Because—as part of Promise Theory—CFEngine agents are also autonomous nodes. Once they have successfully bootstrapped, they will continue to enforce the policy they have. If the server should happen to be down, they'll shrug, say, "Well I don't have a new policy," and keep enforcing what they have. They'll try to reconnect opportunistically of course, but they won't stop working.

As an example, let's say your policy says that root must have a password of `r34lly$ecur3` and the whole network goes down. Sensing opportunity, Bob the Developer (we ALL have a Bob) manages to boot from a CD and set the root password to `b0b0wnz!` on a client because he doesn't like not being root. When that system reboots, it will promptly record that somebody screwed with the password, reset the password, and tell the Policy Server, "Hey, somebody tried to change this password," as soon as it's able. Even if somebody yanks the Ethernet cable and sets the switch on fire.

## So About Those Policies...

There's a reason I keep skipping over policies in detail, and that's because they truly are flexible in CFEngine. Because every possible aspect of a system can be expressed in a policy using domain specific-language (DSL) (and nearly any state as well), there is basically no limit to what you can do in your policies.

Determining what you want to implement through policies in your own network is something only you can decide (and hopefully without the accountants looking over your shoulder). That flexibility, of course, also makes it virtually impossible to really explain or describe what you can do. For every example, there are more than a dozen other ways to do it.

Maybe you want to assign your FreeBSD systems classes based on a regexp against their hostname using a policy—you can do that! Maybe you want to do it statically instead—also possible! The only real limit to what you can do with CFEngine is your imagination (and the time you want to spend writing policies). Simplicity is usually the best answer, not because you can't do it, but because you can easily spend the rest of your life doing nothing but banging away on

one policy.

Because of this power, most folks opt to also implement one of the many excellent third-party promise libraries to ease their workload. The most popular ones are Neil H. Watson's Evolve Thinking library, Normation's `ncf` library, and, of course, the CFEngine Community Open Promise-Body Library (aka `cfengine_stdlib.cf`) that you already installed as part of masterfiles.

All of these libraries are implemented completely in CFEngine's DSL, which makes them OS agnostic. You use the same promise libraries for every system regardless of release and OS. Naturally, there are the usual caveats of CFEngine version compatibility and some functions within those libraries being OS-specific. But because they are written in DSL, they will run on nearly every OS with no changes, and will not run on OSes they do not apply to.

So instead of wondering whether you can do something in policy (you can!), the decision-making process becomes about what should be implemented in policy. Complete and utter madness, right? Making decisions based on what suits your environment instead of what the vendor of the week isn't currently apologizing for not actually delivering? What next, accommodating all those special snowflakes without breaking everything? Oh. Right. I already mentioned you can do that too.

So, let's talk about a practical example: my environment. I run FreeBSD, AIX, Linux, VMware, HyperV, and Windows. It's obviously a complex environment to begin with, made more complex by using NFSv4—so Kerberos is mandatory. I also use automount for home directories, Jenkins, an actual poudriere cluster (no, really!), <Insert Mandatory Cloud Buzzword Here>, and Juniper because—you know why Juniper.

When I stand up a new FreeBSD system, I use a template in VMware that already has `cfengine37` installed. All I do is add the MAC address for `vx0` to my policy that associates it to a hostname. I bootstrap the agent with one command, and CFEngine does all of this for me:

- Kills `dhclient` and installs the correct `/etc/resolv.conf`
- Configures the IPv4 address and default router for `vx0` using DNS
- Configures the IPv6 address and default router for `vx0` using DNS
- Installs the correct pkg repository configuration and updates any packages from base
- Installs and configures the latest version of the support packages I need—`krb5`, `nss-pam-ldapd`, `sasl`, `kstart`, etcetera—that don't belong in the template



- Installs the correct `/etc/krb5.conf` and retrieves the machine's keytab using DNS
- Sets the root password to whatever it is this month (or week, if I forgot it again)
- Installs the correct `autofs` script and enables `autofs` in `/etc/rc.conf`
- Sends me a report telling me that the system successfully bootstrapped and detailing the configuration

What happens after that? That's my baseline policy, so it keeps doing that. If I update the root password promise on the policy server, it will change the root password. If I change the IP address in DNS, it will update `rc.conf` for me. The real magic is that every OS gets the exact same treatment, with the promise adjusted to fit based on the class returned by the agent. (OK, also the real magic is CFEngine on JunOS, but that's between just you and me.)

Because you can combine promises, bundles, and policy into virtually anything, there is really no limit to what you can do with CFEngine. There may not always be an "easy" way to do it, but if you can do it with commands on the OS, you can absolutely do it with CFEngine.

## There Are, of Course, Drawbacks

No software is perfect, and CFEngine is certainly no exception to this. Because it does have more than two decades of history, there are some—shall we say—legacy pieces and behaviors that must be maintained intact for various reasons—usually the reason being someone who is paying a lot of money to CFEngine AS. It's a valid reason—that pays for it to stay open source, you know! But it can cause headaches on modern systems when you run into one.

It's also less a foot-shooting gun, and more a foot-shooting Gatling auto-cannon. There are certainly many ways to minimize the risk. However, eventually, you're going to make a mistake and not catch it. Most of the time, the only impact is the agents refusing to run your new policy because it's broken, and sticking with the previous policy. And sometimes it's `rm -rf /$EmptyVariable/*`—which will get run in parallel across your entire environment, usually in less than 5 minutes.

Because CFEngine is so powerful and flexible, it is also very easy to find yourself buried under the

# RootBSD

## Premier VPS Hosting

RootBSD has multiple datacenter locations, and offers friendly, knowledgeable support staff. Starting at just \$20/mo you are granted access to the latest FreeBSD, full Root Access, and Private Cloud options.



[www.rootbsd.net](http://www.rootbsd.net)

possibilities. Seriously, they're just about limitless. But there are also just as many ways to end up with a sprawling mass of thousands of promises in hundreds of files. I've personally seen setups that rivaled /usr/src—but with a lot less organization. Keeping up with what you need to do and keeping the master files under control can feel like competing interests.

And one of the largest drawbacks by far is that FreeBSD is considered tier 2 by CFEngine. That doesn't mean they don't support it—far from it, thanks to the efforts of the port maintainers and a handful of us users. However, when it comes to function and feature, FreeBSD does not get all the goodies that Linux does. You aren't likely to run into anything you can't do, but cf-agent won't "automagically" report some monitoring data, and you'll have to spend more time writing your own promises and bundles since the standard promise libraries don't prioritize FreeBSD.

## I'd Buy This for a Dollar! Where Can I Learn More?

Thanks to being one of the most mature and stable configuration management systems out there, CFEngine has an absolute mountain of resources available. The best place to start is the official CFEngine site (where you can also grab glossies to slip under the beancounters' doors) at [www.cfengine.com](http://www.cfengine.com). That's also where you can grab CFEngine Enterprise for the price of completely free (for up to 25 hosts). But as with any product this complex and powerful, that's just the beginning.

If you're ready to dive right in, Vertical Sysadmin offers a series of training videos and articles for the low, low price of completely free. With some help from whoever signs the checks, they also provide some of the best in-depth, one-

on-one training you can get. In fact, I would recommend everyone interested in CFEngine start with their IT Automation with CFEngine: Business Values and Basic Concepts video.

However, let's be honest. Getting budget? Right. Not getting budget. For that, CFEngine has the help-cfengine mailing list. As you might expect, you'll see not only CFEngine Champions regularly providing assistance, but also CFEngine developers and employees. Just browsing through the archives, you'll often find the answer to your question is already out there. And there's also the #CFEngine IRC channel on Freenode.

Once you've got your first policy written and you're starting to get comfortable, I highly recommend reviewing the official Best Practices guides before you get too far along. While heavily geared toward Enterprise users, they cover everything from how you should use version control for your policy files to adjusting the policy for scaling to thousands of hosts.

Happy promising! •



**PHILLIP R. JAENKE** is a systems engineer and administrator who also happens to do a bit more than dabble in storage, networking, and writing. He's been at it

long enough to have written checks to Berkeley Software Design. When not busy keeping the lights on at large enterprises across a variety of Unixes, he chips into various open-source projects where he can, including CFEngine and FreeBSD. He also designs and engineers the widely-used BabyDragon VMware reference whitebox, and develops the TaleCaster comprehensive media system.

# SUBSCRIBE TODAY



## FreeBSD<sup>TM</sup> JOURNAL

Go to [www.freebsdjournal.org](http://www.freebsdjournal.org) • 1 yr. \$19.99/Single copies \$6.99 ea.



# AVAILABLE AT YOUR FAVORITE APP STORE NOW



# Puppet

## on FreeBSD

**PRIOR TO PUPPET THE MAIN CONFIGURATION MANAGEMENT TOOL WAS ONE CALLED CFENGINE THAT STARTED IN 1993. PUPPET WAS FIRST RELEASED IN 2005 AND HELPED KICKSTART SOMETHING OF A REVIVAL IN THE CONFIGURATION MANAGEMENT SPACE. NOW IT IS WIDELY CONSIDERED TO BE ONE OF THE MOST POPULAR CONFIGURATION MANAGEMENT TOOLS OUT THERE.**

**By Brad Davis and Andrew Fengler**

---

### Programmatic Configuration

The main strength of Puppet, and any configuration management tool, is the ability to control configuration elements in a programmatic manner. Puppet accomplishes this with an object-oriented system. The most basic element is a resource definition, where a resource could be a file, a package, a command to be executed, or any number of similar things. And if the built-in resource types are not enough, it is possible to define custom resources. Resource definitions along with conditional structures are used to construct classes. These classes are the primary elements giving control on a per-host basis. Each host, or node as Puppet calls them, has a node definition where configuration for that specific host is controlled. Although it is possible to configure all of the resources here, it makes more sense to create classes that can be reused: for instance, an indefinite number of web servers can easily be brought up to an identical state by simply including a class that defines how the web servers should be configured.



## Applying Configurations

There are two ways to run Puppet. The first is to use `puppet apply` to install from a local set of configurations. This is the simplest, but it requires some method of copying the configurations to each node. It also lacks the ability to use PuppetDB. The other method is to run a dedicated Puppetmaster. In this configuration, the nodes will all connect into the central Puppetmaster and fetch the configs from there. The other advantage of this system is that the nodes will send all their facts to the Puppetmaster. Facts are snippets of information about the node, such as the OS, netmask, number of cpu cores, etc. These facts can be used as variables in conditionals allowing control of resources: for instance, it is possible to select a package repository based on the OS version. With all the facts on the Puppet master, a utility such as Puppetboard will provide a view of what's happening on all the servers. It will show breakdowns of facts, the results of Puppet executions, and details on what's changing or has failed. Facts are provided by a library that Puppet depends on called 'Facter'. This tool can be run from the command line to inspect the results on a given machine. The Puppet-specific facts can be viewed by running:

```
# puppet facts
```

## Running on FreeBSD

Puppet, Puppetserver, and PuppetDB can all be installed from the ports tree, making getting started on FreeBSD very simple. For example, to install the current version of Puppet as of the time of this writing, use:

```
# pkg install puppet4
```

Other versions are available to be backwards compatible, as long as they are supported by the Puppet project.

The latest version of PuppetDB can be installed by:

```
# pkg install puppetdb4
```

PuppetDB is not required, but does provide some advanced features to a Puppet installation. For example, the ability to use facts from other nodes on a different node. We will not be covering PuppetDB in this article, but want to mention its availability for advanced users.

From there, simply write a few resource definitions in a class, import that in a node definition, then point the Puppet agent at the server or configurations—depending on which method you picked—and Puppet is up and running! Then comes the fun part of managing everything. There are surprisingly few gotchas for FreeBSD, and as of Puppet 4, pkgng is a supported package provider out of the box. Configuration files can be easily managed by using templating. Puppet uses a templating system from Ruby called ERB, which allows embedding of Ruby code itself into the files and has access to all variables Puppet knows about the environment.

## Example Using agentless

Puppet in so-called agentless mode is a good way to test recipes before applying them to lots of machines. It is very handy to just spin up a jail or VM quickly and run apply to test the configuration.

For example, if we create a file called `'test.pp'` and it contains the following recipe to install nginx:

```
package { 'nginx':  
  ensure => installed,  
}
```

it can easily be executed by running:

```
# puppet apply test.pp
```

For more debugging information to see what might not be working, enabling full debugging mode and verbose mode can be very helpful:

```
# puppet apply -d -v test.pp
```

## Example Using a master/agent

After installing Puppet, on the master, enable it with:

```
# sysrc puppet_master_enable=YES
```

Then start up the master:

```
# service puppet_master start
```

On the client (which could be the same machine):

```
# sysrc puppet_enable=YES
```

Modify the puppet.conf to point to the IP or hostname of the master. In this case, we will be pointing to localhost:

```
master: 127.0.0.1
```

And start up the client:

```
# service puppet start
```

Once the master and agent are started, the agent will attempt to connect to the master. They will then create an internal Certificate Authority to authenticate each other. The agent will generate a client certificate and upload it to the master. You will be able to see the agent waiting for the master to accept its certificate in the logs. This certificate can be viewed on the master by running:

```
# puppet ca list
```

Then you should see the certificate from your client. It will look like:

```
"hostname" [sha256 checksum of certificate request]
```

Now sign the certificate:

```
# puppet ca sign hostname
```

The agent should receive the certificate and continue on with its run. However, it's not going to do very much, and will spit out a big yellow warning:

```
ERROR: Could not fetch my node definition, but the Puppet agent run will continue
Looks like we'll need to create a node definition.
```

## Creating a Simple Configuration

Let's start by going to our Puppet configuration directory, `/usr/local/etc/puppet/` by default. Our configuration environments go in the "environments" directory. You can have multiple environments, and the Puppet agent can be configured to select which environment it uses. This means you can have separate branches for development and so on, but, for now, the default "production" environment should be perfect for testing. Each environment has a "manifests" and a "modules" directory. Site-wide manifests, such as common configuration and node definitions, go under manifests, and modules holds reusable blocks of configuration.

Puppet files end in `.pp`, so let's create our node definition. A node definition matches the hostname of the node, either exactly, or a regex match. Create `localhost.pp` with the following content:

```
node 'localhost.example.com' {
    $host_desc = "My computer"
    $colorscheme = "desert"

    file { ["/etc/test":
        ensure => present,
        user   => "root",
        group  => "wheel",
        mode   => "0644",
        content => "Hello World! This is ${host_desc}\n",
    ]
}
```

And that's a simple node definition. We set two variables to use later and created your first file. You can see that the file has a number of options set, namely its owner, group, permissions (mode), and what goes in the file. Save this, and do another Puppet run. If you started the Puppet service, it will run every so often, but if you don't want to wait, you can trigger a run with the `--test` or `-t` flag:

```
# puppet agent -t
```

And that will set off a run. When it's finished, you'll have a shiny new file.

Now let's create our first module. Let's say we want our favorite editor and all its settings on our computer. Under the modules directory, create a directory called "vim". Inside that, a number of subdirectories are used to hold different parts of the module. Create a "manifests" directory; this will hold all the Puppet configuration for this class. For modules, there must always be a class that has the same name as the module in a file called "init.pp". This is the starting point of the module, and we can add more files with extra classes to include if needed. Create init.pp:

```
class vim {
  package { "vim-lite":
    ensure => latest,
    name   => "editors/vim-lite",
    provider => "pkgng",
  }
  file { ["/home/beastie/.vimrc":
    ensure => present,
    user   => "beastie",
    group  => "beastie",
    mode   => "0644",
    source => "puppet:///modules/vim/vimrc",
  ]
}
```

We've managed two more resources: a package that will install vim-lite using pkgng, and a file that will install our vimrc. Notice how we used "source" instead of "content". Instead of placing some text in a file, this will get a file from the Puppetmaster. These files also go in the "vim" module, but under a "files" subdirectory instead of in the manifests directory. The format for the file URLs is "puppet:///modules/<module name>/<file name>". So, our "puppet:///modules/vim/vimrc" URL will look for "modules/vim/files/vimrc" under the current environment.

Every resource type offers a different set of options. For files, there are a wide variety of options, one of which is "ensure". This determines what state we want the file to be in. When we're done with that file, you can clean it up by setting ensure to "absent".

Now that we have a class, just add the line "include vim" to our node definition, and it will add the entire vim class. We can add it to as many nodes as we want with just a single line. But let's say we need to change a setting on some servers. This is easy to do with a template.

## Templating

Templating is a very powerful feature in any automation system. The templating system has access to all the variables from Facter, so it knows all about the machine. It allows for very complex configuration across many machines to be expressed simply. For example, to template a configuration file where it is necessary to do something like set the IP address to bind to, first create a resource for the file like:

```
file { ["/usr/local/etc/nginx/nginx.conf":
  ensure => 'file',
  owner  => 'root',
  group  => 'wheel',
  mode   => '644',
  content => template('/usr/local/etc/puppet/templates/nginx.conf'),
  require => Package['nginx'],
]
```



The templating engine works on template code found between "<%= " and "%>". Now here is a snippet of what the template could contain:

```
http {
    server {
        listen <%= @ipaddress %>:80;
        server_name <%= @fqdn %>;
        location / {
            root /usr/local/www/nginx;
            index index.html;
        }
    }
}
```

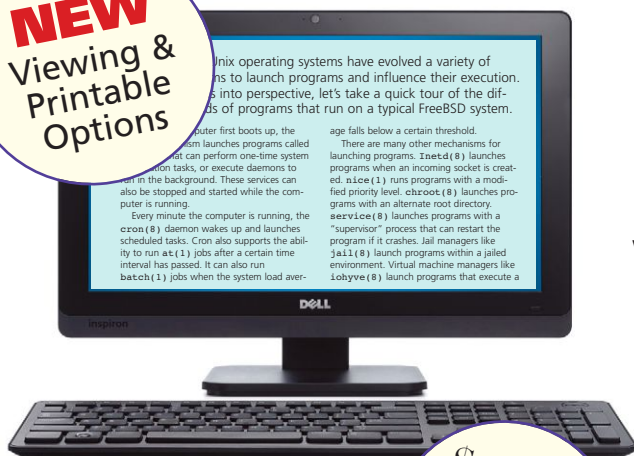
Note that the variables begin with an "@" symbol. This will give us a result like:

```
http {
    server {
        listen 192.168.11.10:80;
        server_name test.example.com;
        location / {
            root /usr/local/www/nginx;
            index index.html;
        }
    }
}
```

There are many more powerful things that can be done with templating including using Ruby functions to create more advanced functionality. As a slightly more complicated example of what can be done with templating, one time it was used to modify one of the variables using the Ruby gsub function



**NEW**  
Viewing &  
Printable  
Options



The DE, like the App, is an individual product. You will get an email notification each time an issue is released.

**\$19.99**  
YEAR SUB  
**\$6.99**  
SINGLE COPY

## The Browser-based Edition (DE) is now available for viewing as a PDF with printable option.

The Browser-based DE format offers subscribers the same features as the App, but permits viewing the *Journal* through your favorite browser. Unlike the Apps, you can view the DE as PDF pages and print them.

**To order a subscription, or get back issues, and other Foundation interests, go to**

**[www.freebsdoundation.org](http://www.freebsdoundation.org)**

and then the result was embedded in the file.

## Dependencies

One issue you can run into is that some resources will require other resources in order to function. Starting a service before you have its config file in place will not likely do what you want. Dependencies are a lot better in Puppet 4, as resources are now applied in the order you place them in files. Previously, this was not the case as they were sorted prior to execution. However, you may still need to specify a dependency order.

The primary way is with the `"require"` keyword. There are two ways to use it. You can require an entire class:

```
require nginx
```

This works exactly like `"include nginx"`, but will ensure the entire class is applied before continuing. The other way is to require another resource:

```
package { 'zsh':
  ensure => 'installed',
}
user { "beastie":
  ensure  => "present",
  comment => "Beastie",
  groups  => ["beastie", "wheel"],
  shell   => "/usr/local/bin/zsh",
  require => Package['zsh'],
}
file { "/home/beastie":
  ensure  => "directory",
  mode    => "0644",
  owner   => "beastie",
  group   => "beastie",
}
```

This user will depend on the Z Shell package being installed, and because the files depend on the owner and group, those will automatically depend on the user.

## Node Interaction

You may need servers to be aware of each other, whether to whitelist IPs, or to know where to listen for a heartbeat. The difficulty is that normally this would require hardcoding long lists of addresses. Puppet has a feature called "exported resources". These resources are much like normal resources, but instead of being installed on the host, they are stored on the Puppetmaster for any node to collect. Resources can also be tagged, making it easy for a server to pick up an entire directory of configs with a single line.

We can export a file like this:

```
@file { "/usr/local/etc/nagios/puppet.d/jail_${hostname}.dns.cfg":
  ensure  => present,
  content => template('nagios/server.cfg.erb'),
  tag     => "nagios_config",
}
```

And then pick up any files tagged `"nagios_config"` on another host:

```
File <<| tag == "nagios_config" |>> {
}
```

This will create the file on the server that picks it up. Note that we use the hostname in the file-name. This way if we export configs from several hosts, they won't collide.

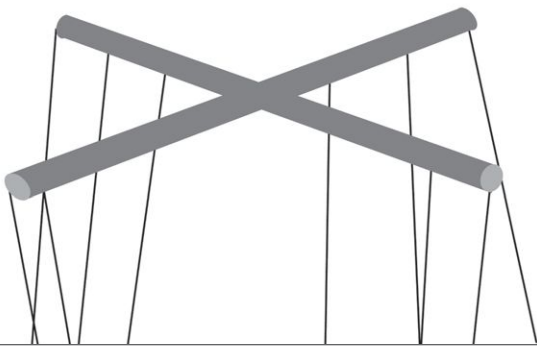
## Conclusion

There are a few different ways to view the documentation, but one of the best sources is the Puppet documentation website available here:

<https://docs.puppet.com/puppet/latest/>. Additionally, for help for specific Puppet commands, from the cli:

```
puppet help <cmd>
```

This provides a good foundation to getting started with Puppet and making life easier for any Systems Administrator or anyone else interested in automating building of their environments. •



**ANDREW FENGLER** is a Unix administrator working for ScaleEngine Inc. in Hamilton, Canada, where he manages a large number of Unixen with Puppet. He has two years' experience administering FreeBSD systems at scale.

Professionally, **BRAD DAVIS** has been infrastructure systems architect, developer, and is now a consultant. He has been a FreeBSD committer for over 10 years and involved in many different areas of the Project. Initially starting out as a documentation committer, he used that to launch into helping the Cluster Administration and Postmaster teams. After retiring from those teams, he dabbled in pkg and poudriere. These days he is working on various ports, packaging the FreeBSD base system and a project called RaspBSD to nicely package up FreeBSD and some extra tools for ARM boards like the BeagleBone Black and the Raspberry Pi.

Let **FreeBSD Journal** connect you with a targeted audience!

**Advertise Here**

# CLIMB WITH US!

→ **LOOKING** for qualified job applicants?

→ **SELLING** products or services?

Email [walter@freebsdjournal.com](mailto:walter@freebsdjournal.com)  
OR CALL



## 888/290-9469



BY  
STEVE WILLS

Vagrant (<https://www.vagrantup.com/>) is a command-line utility for building and managing portable development environments such as virtual machines (VMs). It creates, starts up, provisions, and destroys VMs easily. It is often used as a tool for setting up development and test environments such as on a laptop.

# VAGRANT

Vagrant allows you to set up these environments in an automated and reproducible way, which ultimately increases productivity and allows you to share these automated environments with others. Vagrant, written in Ruby (<https://www.ruby-lang.org/>), is a cross-platform utility. It easily runs on Mac OS, Windows, Linux, and FreeBSD. By supporting multiple hypervisors—including VirtualBox, VMWare, Bhyve (with some work)—as well cloud hosting—such as Amazon AWS, Google Cloud Compute, Azure, and many others—Vagrant offers a lot of flexibility. In terms of provisioning or initial VM setup, it supports a variety of tools, such as Ansible, Chef, Puppet, Salt, and even shell scripts.

Hypervisor and provisioner support is provided by plugins, so end users can easily develop plugins for particular providers or provisioners. There are many plugins available (<https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins>).

## Why Use Vagrant?

You can create Vagrant environments and easily share them with your coworkers regardless of which OS they use on their laptops or desktops. And you can bring up a single or multi-host setup with a single `vagrant up` command. By making development environments reproducible, you can avoid the common "works on my machine" type of problem.

Vagrant is typically used on laptops or desktop workstations for setting up development and test environments, but it can be used elsewhere. For example, one interesting use case is for creating build nodes or temporary hosts for automated build environments or tools such as Jenkins. Generally, Vagrant is used by software developers or testers. It can be useful for operations folks, too, such as for modeling production environment setups.

Vagrant is a very useful tool meant for reproducible, ephemeral development and testing environments. However, it is not meant for managing production hosts with critical data. Vagrant's simple design makes it far too easy to destroy everything you have created with it for that type of usage.

## How to Get Vagrant

You can install Vagrant on FreeBSD using the command `pkg install vagrant`. This will install Vagrant and all of its dependencies. You will also want to install a hypervisor. For the purposes of this article, I will demonstrate both VirtualBox and Bhyve as hypervisors. To install VirtualBox on FreeBSD, use `pkg install virtualbox-ose`. Be certain to follow any instructions about system setup that are listed.

## Components of a Vagrant Project

There are two main components to Vagrant environments. The first is what is called the Vagrant "box". This is the wrapper of a disk image containing an installed guest OS and meta-data about that disk image and OS. Users can create these themselves—manually—and share them or use pre-created boxes created by other users or groups. Prebuilt boxes can be found at <https://atlas.hashicorp.com/boxes/search>. The `vagrant` command will search this location by default for boxes that do not already exist locally. Boxes can be manually added to the local Vagrant cache using the `vagrant box add` command,

such as in the following example:

```
vagrant box add hashicorp/precise64
```

You can list cached boxes with

```
vagrant box list.
```

The second main component is the **Vagrantfile**. This file lists the VMs that are in your Vagrant project and how to set up those VMs. While the Vagrantfile is written in Ruby, editing it does not require extensive knowledge of Ruby. It is intended to be simple. More information on the **Vagrantfile** is available at <https://www.vagrantup.com/docs/vagrantfile/>.

The Vagrantfile also serves to mark for Vagrant where the root of your project lives. This helps Vagrant find other files that may be referenced in the Vagrantfile using relative file reference.

## Setting up a Project

The simplest way to set up a project is to run the following command:

```
vagrant init hashicorp/precise64
```

This will create the Vagrantfile in the current directory. Next run:

```
vagrant up
```

This command will download the box to a local directory (`~/.vagrant.d/boxes/` by default) and start up the VM. After this, you can run:

```
vagrant ssh
```

to ssh into the VM and begin using it. When you are done, you can stop the VM using:

```
vagrant halt
```

Or if you want to get rid of the VM completely, run:

```
vagrant destroy
```

This will properly remove the instance of the VM with its local changes, but leave the cached copy of the box in `~/.vagrant.d/boxes/`.

In our example VM, notice we did not specify any provisioning or configuration steps. We can add steps to the Vagrantfile to configure the VM. For example, uncomment these lines in the

Vagrantfile:

```
config.vm.provision "shell", inline: <<-SHELL
    apt-get update
    apt-get install -y apache2
SHELL
```

The next time you run `vagrant up` or `vagrant provision`, Vagrant will run the `apt-get` commands to install the Apache web server.

More information on provisioning is available at <https://www.vagrantup.com/docs/getting-started/provisioning.html>.

One interesting feature of Vagrant is the synced folder feature. It will sync files to and from the host operating system to the guest operating system, making it easier, for example, to edit code you want to build or run in the VM. The VM has a default network connection that allows connecting to the VM with `ssh`. You can add additional port forwards to the VM on any additional ports needed.

## Creating a New Vagrant Project

As an example, we will create a new Vagrant project running a web server. The first step of creating a new Vagrant project is creating a new directory and running `vagrant init` in that directory. For example:

```
mkdir journal
cd journal
vagrant init freebsd/FreeBSD-11.0-RELEASE-p1
```

This will create a Vagrantfile in our new directory. Note that we will want to make a number of changes to this default file before using it. Due to defaults of Vagrant, you will want to add the following line to our new Vagrantfile:

```
config.vm.guest = :freebsd
```

Also, due to a bug in the FreeBSD box (for which a fix is available and will be committed soon), you will need to add the following line to the new Vagrantfile:

```
config.vm.base_mac = "080027FC8C33"
```

To avoid a boot timeout during the first boot, as the system performs its first boot update, we need to add the following line to our Vagrantfile:

```
config.vm.boot_timeout = 3600
```

There is another change to make before we can try to initialize our box, as we need to disable the default shared folders:

```
config.vm.synced_folder ".", "/vagrant", id: "vagrant-root", disabled: true
```

Also, we'll want to expose a port for our web server by adding this line:

```
config.vm.network :forwarded_port, guest: 80, host: 8080
```

And, since we do not have `bash` installed by default on FreeBSD, we need to configure Vagrant to use `sh` as its `ssh` shell:

```
config.ssh.shell = "sh"
```

The default amount of RAM specified is 512MB, which we should increase to 1GB. And we will allocate two CPUs to the VM:



```

config.vm.provider :virtualbox do |vb|
  vb.customize ["modifyvm", :id, "--memory", "1024"]
  vb.customize ["modifyvm", :id, "--cpus", "2"]
end

config.vm.provider :bhyve do |vm|
  vm.memory = "1024M"
  vm.cpus = "2"
end

```

Notice how we have separate blocks for each hypervisor provider. Let's also add additional disk to the VM for use with ZFS. This will be hypervisor specific as well. In the section for virtualbox, we add:

```

file_to_disk = File.realpath( "." ).to_s + "/extradisk.vdi"

if ARGV[0] == "up" && ! File.exist?(file_to_disk)
  vb.customize [
    'createhd',
    '--filename', file_to_disk,
    '--format', 'VDI',
    '--size', 3 * 1024 # 3 GB
  ]
end
vb.customize [
  'storageattach', :id,
  '--storagectl', 'IDE Controller', # The name may vary
  '--port', 1, '--device', 0,
  '--type', 'hdd', '--medium',
  file_to_disk
]

```

And in the section for bhyve, we add:

```

vm.disks = [{name: "extradisk", size: "3G", format:"raw",}]

```

Our next step is to install the web server process and ensure that it is running.

```

config.vm.provision "shell", inline: <<-SHELL
  pkg install -y apache24
  sysrc apache24_enable=yes
  service apache24 start
SHELL

```

So, our Vagrantfile, minus comments, should look like the following:

```

Vagrant.configure("2") do |config|
  config.vm.box = "freebsd/FreeBSD-11.0-RELEASE-p1"
  config.vm.guest = :freebsd
  config.vm.base_mac = "080027FC8C33"
  config.vm.boot_timeout = 3600
  config.vm.network :forwarded_port, guest: 80, host: 8080
  config.vm.synced_folder ".", "/vagrant", id: "vagrant-root", disabled: true
  config.ssh.shell = "sh"

  config.vm.provider :virtualbox do |vb|
    vb.customize ["modifyvm", :id, "--memory", "1024"]
    vb.customize ["modifyvm", :id, "--cpus", "2"]
    file_to_disk = File.realpath( "." ).to_s + "/extradisk.vdi"

```

```

if ARGV[0] == "up" && ! File.exist?(file_to_disk)
  vb.customize [
    'createhd',
    '--filename', file_to_disk,
    '--format', 'VDI',
    '--size', 3 * 1024 # 3 GB
  ]
end
vb.customize [
  'storageattach', :id,
  '--storagectl', 'IDE Controller', # The name may vary
  '--port', 1, '--device', 0,
  '--type', 'hdd', '--medium',
  file_to_disk
]
end

config.vm.provider :bhyve do |vm|
  vm.memory = "1024M"
  vm.cpus = "2"
  vm.disks = [{name: "extradisk", size: "3G", format:"raw",}]
end

config.vm.provision "shell", inline: <<-SHELL
  pkg install -y apache24
  sysrc apache24_enable=yes
  service apache24 start
SHELL
end

```

Now we can start our VM using `vagrant up`. When Vagrant creates and starts our VM, it creates it as a headless VM that is not visible. You may want to open the VirtualBox Manager so that you can see the VM in the VM list. You can also click the VM and then click the "Show" button to see the VM console. While the shell provisioning script is running, we can view the output of the commands. After this finishes, we should be able to bring up the default website:

```
http://localhost:8080/
```

To access the box, you can `ssh` in using the previously mentioned `vagrant ssh` command.

## Existing Projects

One of the strongest powers of Vagrant comes from using the automation to create boxes and Vagrantfiles and sharing them with other users. So, let's look at some existing projects. Note these examples were tested with the VirtualBox hypervisor.

### Minio

Minio (<https://minio.io/>) is a "distributed object storage server built for cloud applications and devops." It is compatible with Amazon's S3 and can be pretty useful. For this example, I have created a Vagrantfile that uses Ansible to configure Minio in FreeBSD. First, install Ansible by running `pkg install ansible`; then clone or download this project (<https://github.com/swills/minio.vagrant>). You can then run the `vagrant up` command and have a working Minio server with just a few simple commands.

```

mkdir minio
cd minio
git clone https://github.com/swills/minio.vagrant.git .
vagrant up

```

The `vagrant up` step will take a few minutes, as the FreeBSD VM does an automated `freebsd-update` on first boot. After it finishes, `ssh` into the VM and get the authorization keys for Minio by running `vagrant ssh`. Then in the VM, execute the following commands:

```
sudo grep accessKey /usr/local/etc/minio/.minio/config.json
"accessKey": "KO2E80M9H87CDIJKP6A1",
sudo grep secretKey /usr/local/etc/minio/.minio/config.json
"secretKey": "2mMkAC0oq7v8oUrZqZs3+S9gRwqNLws5MZTNjgnE"
```

You can then login to Minio at:

```
http://localhost:9000/minio/login
```

You will need to use the `accessKey` and `secretKey` found in the VM. (These are generated at the time the service is first started and will vary.)

## Gluster

GlusterFS ([http://www.gluster.org/documentation/About\\_Gluster/](http://www.gluster.org/documentation/About_Gluster/)) is a free software parallel distributed filesystem, capable of scaling to several petabytes. Similar to the previous example, I've created a Vagrantfile that uses Ansible to configure Gluster on a pair of FreeBSD VMs. You can clone this project (<https://github.com/swills/okapi>) and run `vagrant up` to get started.

```
mkdir gluster
cd gluster
git clone https://github.com/swills/okapi.git .
vagrant up
```

Next, run the `vagrant ssh server1` command. Once you are in that VM, run the following command:

```
sudo gluster peer probe server2
sudo gluster volume create gv0 replica 2 server1:/gluster/gv0 server2:/gluster/gv0
sudo gluster volume start gv0
sudo gluster volume info
sudo mount_glusterfs server1:/gv0 /mnt
```

That will set up Gluster. Then you can `vagrant ssh server2` and in that VM run:

```
sudo mount_glusterfs server1:/gv0 /mnt
```

From this point, you should have a working Gluster cluster. Any file created or modified in `/mnt` on one should be available in the other. (The `gluster` commands are run manually so that they happen after both machines are up.)

## Poudriere

Finally, and perhaps of particular interest to anyone interested in doing FreeBSD ports development, there is a Vagrantfile and set of Ansible scripts that will set up a complete `poudriere` development environment. This includes a complete ports tree for doing ports development and multiple jails for testing builds. Note that setting up this one will take significantly longer to provision and will require more disk space due to the jail build process and the ports tree checkout.

```
mkdir poudriere_vagrant
cd poudriere_vagrant
git clone https://github.com/swills/gerenuk.git .
vagrant up
```

After this finishes successfully, you can `vagrant ssh` into the VM and work with ports and run `poudriere` to test builds. You can edit files in `/usr/local/poudriere/ports/default` and build them using standard `poudriere` commands, for example:

```
sudo poudriere bulk -j 110-amd64 shells/zsh
```

The provisioning scripts have also set up NGINX to serve the `poudriere` statuspages at:

```
http://localhost:10080/
```

## Using Vagrant with Bhyve

Vagrant can be used with Bhyve via a project called `vagrant-bhyve` (<https://github.com/jesa7955/vagrant-bhyve>) that was created during the Google Summer of Code project in 2016. In order to use it, you will probably want another project that helps convert VMs to formats that `vagrant-bhyve` can understand, called `vagrant-mutate` (<https://github.com/sciurus/vagrant-mutate>). You can install both on FreeBSD by running:

```
pkg install rubygem-vagrant-bhyve rubygem-vagrant-mutate
```

After installing the plugin as we did above, we need to ensure `vagrant` is using it by running the following commands:

```
vagrant plugin install vagrant-mutate
vagrant plugin install vagrant-bhyve
```

As mentioned previously, we can list currently locally cached boxes with the `vagrant box list` command. Our output might look like this:

```
freebsd/FreeBSD-11.0-RELEASE-p1 (virtualbox, 2016.09.29)
hashicorp/precise64 (virtualbox, 1.1.0)
```

We can use `vagrant mutate` to convert these boxes for use with Bhyve. To convert the box, use the following commands:

```
vagrant mutate ubuntu/xenial64 bhyve
vagrant mutate freebsd/FreeBSD-11.0-RELEASE-p1 bhyve
```

We can now create a VM using our previously generated journal Vagrantfile by doing the following:

```
vagrant up --provider bhyve
```

This will, as always, update on first boot, so it may take some time. Due to the automatic reboot after update, the VM will shut down and Vagrant will indicate that the VM failed to come up, but if you rerun the `vagrant up --provider bhyve` command, it should work fine. After this, you can run the provisioning step by running `vagrant provision`. And then, just as when we were using VirtualBox, `ssh` in via `vagrant ssh`. The `vagrant-bhyve` plugin does not currently support setting up the port forwarding, so that will have to be done manually in this case. You can view the console of the VM by running the `cu -l /dev/nmdm0B` command.

## Conclusion

I hope you enjoyed this brief tour of Vagrant, and, when you try it out, I hope you find it as useful as I do!

---

**STEVE WILLS is a husband and father living in North Carolina. He is a FreeBSD ports committer with a focus on Ruby and other programming languages.**







**Testers, Systems Administrators,  
Authors, Advocates, and of course  
Programmers** *to join any of our diverse teams.*

# COME JOIN THE PROJECT THAT MAKES THE INTERNET GO!

★ **DOWNLOAD OUR SOFTWARE** ★

<http://www.freebsd.org/where.html>

★ **JOIN OUR MAILING LISTS** ★

<http://www.freebsd.org/community/maillinglists.html?>

★ **ATTEND A CONFERENCE** ★

**OSCON 2017** • Austin, Texas • May 8–11, 2017

**Rootconf 2017** • Bangalore, India • May 11–12, 2017

**BSDCan 2017** • Ottawa, Ontario, Canada • June 7–10, 2017

The FreeBSD Project

  
**FreeBSD**  
FOUNDATION

# A PRIMER

By Peter Wright

Configuration management comes in many flavors and philosophies. From simple Makefile-based automation tasks to fully orchestrated deployment and administrative environments, there are a multitude of approaches one can take.

In this article, we will take a look at SaltStack, a Python-based configuration management solution that is feature rich and has a reputation for ease of use and scalability. SaltStack also does a very good job in abstracting various OS-specific implementation details from administrator; for example, the same syntax is used to install native OS packages on GNU/Linux as well as on FreeBSD. This makes the task of managing heterogeneous environments much easier, while also encouraging code reuse when possible, regardless of the plat-

form being targeted.

SaltStack is also designed to support large-scale installations. This is primarily due to the architectural decision of using a publish-subscribe messaging model where the Master does not have to craft a configuration directive for each node managed; rather, it can broadcast directives on a channel that clients (Minions in Salt parlance) subscribe to, and, accordingly if the directive applies to them. Communication via the pub/sub method is also encrypted, with user-definable key rotation.

## SaltStack on FreeBSD

Another feature that SaltStack has implemented that is beneficial for larger installations is what is called event-driven infrastructure (EDI). This allows Salt clients to react to changes in local system state, or events external to SaltStack itself. This is a popular feature for cloud-based infrastructure where capacity is managed dynamically in relation to load on the overall system.

This article is intended as a primer and will take a quick tour of the SaltStack architecture, illustrating that the barrier of entry to start using it is actually quite low. We will also walk through a real-world example of configuring a FreeBSD-based Redis server, then finally highlight some interesting features that I encourage readers to investigate on their own.

## Architecture Overview

The architecture of SaltStack is relatively straightforward, yet, due to several design decisions, it is actually quite scalable and easy to customize to suit the specific needs of a given environment. The primary node in a SaltStack cluster is called a Master, and client systems are referred to as Minions. The Master is responsible for publishing messages for the Minions. The Master also manages who is a valid member of a given cluster by accepting an initial key from a Minion, then keeping track of new keys from Minions as they are automatically rotated. In regard to Master/Minion communication, SaltStack uses ZeroMQ as the foundation for its pub/sub message queue.

SaltStack also has the concept of a Grain, akin to Facter in Puppet, which stores system variables such as the name of your OS, the amount of cores available, and what a given Minion's primary IP address is. This is very helpful when developing templates, and is something that will be illustrated below. SaltStack also has the concept of a Pillar that is used to store user-defined variables such as file paths or even passwords. Templates in SaltStack are achieved by using Jinja, and can be used to insert logic into your SaltStack configurations. It is common for them to leverage both Grains and Pillars, allowing for greater flexibility and more succinct configuration directives.

## Real World Example

With this background, let's use a real world example of setting up the Redis key/value datastore on a FreeBSD server using SaltStack. First we will set up a SaltStack Master and Minion; then we will walk through a simple SaltStack "statefile" which will deploy a basic Redis configuration.

## Preparing a Host to Use SaltStack

Installing SaltStack is straightforward using pkg(7); in fact, the same package is used for installing a SaltStack Master or Minion.

```
% sudo pkg install py27-salt
```

We will skip configuring the salt\_master and salt\_minion daemons, as that is covered quite well by the SaltStack documentation. After configuring both systems and successfully starting the salt\_master daemon, you can then start the salt\_minion, which will connect to the Master sending over its initial public key for acceptance. You can view and accept the key using the "salt-key" command on your Master like so:

```
$ sudo salt-key
Accepted Keys:
Denied Keys:
Unaccepted Keys:
test0.com.puter
Rejected Keys:
```

Above, we can see that we have a pending key from "test0.com.puter". Before accepting the key, you can run "salt-key -f test0.com.puter" on the Master to view and confirm the fingerprint of this key matches the fingerprint of the Minion. Once you have verified the key, you can now accept it like so:

## Getting Comfortable with SaltStack Commands

```
$ sudo salt-key -a test0.com.puter
The following keys are going to be accepted:
Unaccepted Keys:
test0.com.puter
Proceed? [n/Y] y
Key for minion test0.com.puter accepted.
```

At this point, we now have a Minion securely paired with our Master, and we can verify this by using a function built into SaltStack that allows for arbitrary commands to be executed. Here we instruct the Master to publish a message stating all hosts matching the test\* regular expression to run the "hostname" command. Any systems matching the regex will execute the argument to cmd.run and will post its output back on the message queue for the Master to consume:

```
$ sudo salt 'test*' cmd.run 'hostname'
test0.com.puter:
    test0.com.puter
```

---

#### Summary

---

```
# of minions targeted: 1
# of minions returned: 1
# of minions that did not return: 0
# of minions with errors: 0
```

---

This example actually tells us several other interesting things about SaltStack. Firstly, the Salt command accepts regular expressions as inputs for hosts to target commands against. So, if I were to have a cluster of systems with a common element in their hostnames, I could succinctly target all of them via the Salt CLI tool. Secondly, Salt supports running ad-hoc commands against registered Minions. For example, using this functionality, I can execute an ad-hoc query to verify all my web-servers have the appropriate Apache 2.4 package installed. But let's take a look at a more traditional use of SaltStack, applying a configuration state (called a SaltState) to our new node.

## Using SaltState Files to Manage Minions

SaltStack configuration directives are stored in what are called statefiles denoted with the `.sls` extension. Each SaltStack installation has what is called a "top file" that defines the overall structure and association of SaltStates to hosts under management. For our purposes we will start with a very simple top file that associates two states to our node like so:

```
1 base:
2   '*':
3     - motd
4
5 dev:
6   'test*':
7     - redis_demo
8
```

SaltStack statefiles follow standard YAML syntax and hierarchy rules. In our example above, lines 1–3 define a base class that matches all systems associating a salt state file named "motd". The motd statefile looks like this:

```
1 {% if grains['os'] == 'FreeBSD' %}
2 /etc/motd:
3   file.managed:
4     - source:salt://global_files/motd_freebsd
5     - user: root
6     - group: wheel
7     - mode: 644
8 {% endif %}
```

This example statefile actually has embedded some Jinja templating code inside it, which, in our case, determines if the host we are running on is FreeBSD, and if so, we apply the "motd\_freebsd" file to the host with the defined ownership and permissions. The Minion executing the statefile will search its grain inventory checking to see if the value of the "os" key matches FreeBSD. If it matches, we copy over our FreeBSD motd file to /etc/motd, ensuring its ownership and mode.

You can also interact with Grains using the SaltStack CLI like so:

```
$ sudo salt 'test*' grains.get os
test0.iad0.tribdev.com:
    FreeBSD
```

---

#### Summary

---

```
# of minions targeted: 1
# of minions returned: 1
# of minions that did not return: 0
# of minions with errors: 0
```

---

```
$
```

The above command, when invoked on the Master, targets all configured Minions whose name matches the 'test\*' regular expression. You can view all available key/value pairs for a given node by substituting "grains.get" with "grains.items".

Being able to embed templating and logic inside a state configuration file is powerful; but SaltStack also allows you to use templating for file resources, which we'll cover next.

## Detour into Environment Separation

This is a good time to take a quick tour of how SaltStack organizes files on disk. On my Master, my directory structure of statefiles and file resources looks like so: (next page)



```
$ cd $SALT_ROOT
$ find .
./states/top.sls
./states/global_files/motd_freebsd
./states/dev/states/redis_demo.sls
./states/dev/states/dev_files/redis_demo.conf
./states/dev/states/dev_files/redis_rc
```

One thing you should notice is that our top.sls file is in a separate location as the redis\_demo.sls file; redis\_demo.sls is a child of the dev/states directory hierarchy. What I have done here is create a separation of environments using this structure. Specifically, the root level contains the top.sls file and our motd\_freebsd file. States in this location are all hosts as they have a global scope. Then I have created a “dev” directory structure that contains our redis statefile and supporting file resources. This dev environment is reflected in the previous top file via lines 5–7, and as you’ll see in line 6, we are more restrictive in the hosts that are matched via our regular expression. You can also have an arbitrary number of environments in

SaltStack, which is very handy. For example, I have several development environments in addition to a pre-production and production environment at my larger SaltStack installation. This allows me to federate who has access to which environments while also helping me ensure changes done in one environment do not affect other environments.

## A More Complex Statefile

Now that we’ve broken down environments and SaltStack filesystem layouts, let’s take a look at our redis\_demo.sls file that installs several packages and references a templated file resource:

# FreeBSD JOURNAL



## DID YOU MISS?



- Nov/Dec 2015 / **Olivier Cochard-Labbé**, **The BSD Router Project**
- Jan/Feb 2016 / **Peter Holm**, **Using Fuzzy Testing to Build Industrial-Strength Systems**
- March/April 2016 / **Brooks Davis**, **Cheri**
- May/June 2016 / **Andy Waafa**, **ARMv8**
- July/Aug 2016 / **Chris Johns et al**, **FreeBSD and RTEMS**
- July/Aug 2016 / **Michael Lucas**, **Tuning ZFS**

## Get caught up today

Order Back Issues @ [www.freebsd.foundation.org/journal](http://www.freebsd.foundation.org/journal)

```

1 /usr/local/etc/redis.conf:
2   file.managed:
3     - require:
4     - pkg: redis_pkgs
5     - source: salt://dev_files/redis_demo.template
6     - template: jinja
7     - user: root
8     - group: wheel
9     - mode: 644
10
11 /var/db/redis/:
12   file.directory:
13     - user: redis
14     - group: redis
15     - mode: 755
16
17 /etc/rc.conf.d/redis:
18   file.managed:
19     - source: salt://dev_files/redis_rc
20     - user: root
21     - group: wheel
22     - mode: 644
23
24 redis_pkgs:
25   pkg.installed:
26     - pkgs:
27       - redis

```

Lines 1–9 define the characteristics of a file deployed to `/usr/local/etc/redis.conf`. Lines 3–4 create a dependency on the packages defined in lines 24–27 to be installed before this file resource can be deployed. Next, we reference a Jinja template that is our main Redis configuration file at line 5. Finally, through lines 11–22, we ensure a directory exists, with correct permissions for Redis to checkpoint data to disk; and we also ensure the Redis daemon is enabled via rc.

Let's take a look at the `redis_demo.template`, as that will further illustrate how you can leverage SaltStack Grains and templating to write one configuration file that can be deployed on multiple systems:

```

1 {% set pri_ipv4 = grains['ipv4'][0] %}
2
3 protected-mode no
4 port 6379
5 tcp-backlog 511
6 bind {{ pri_ipv4 }}
7 timeout 0
8
9 tcp-keepalive 300

```

The above snippet represents the first nine lines of a Redis configuration file; it illustrates

how Grains and templates can be combined in a pretty powerful manner.

The first line defines a variable “`pri_ipv4`” and populates it with the first value present in the `ipv4` key as reported by the SaltStack Grains system. To view what that Grain will look like in a shell you can execute the following:

```

$ sudo salt 'test*' grains.get ipv4
test0.iad0.tribdev.com:
  - 10.3.16.51
  - 127.0.0.1

```

---

#### Summary

---

```

# of minions targeted: 1
# of minions returned: 1
# of minions that did not return: 0
# of minions with errors: 0

```

---

SaltStack returns these values as a list, and, as such, we have to specify the element in the list that represents the value we would like to use. So, when the Minion evaluates this salt stage, it scans the grain system for a key named “`ipv4`” and sets the `pri_ipv4` variable with the public IPv4 address that is stored. This data is then inserted into the local configuration file on the Minion as per line 6, where “`{{ pri_ipv4 }}`” is substituted with the IP addresses.

This is a very simple example, but the power should be pretty evident. For example, imagine you manage a fleet of systems spanning several sub-domains. Using Grains and templates, you can abstract many of the per-domain and system configurations in your template and even use logic to determine which parts of a template are rendered on a given Minion. SaltStack also makes it trivial to extend the default Minions by writing simple Python classes.

## Applying SaltState Configurations

Now that we have gone through our statefiles, it's time to apply them to our Minion. We will do this by executing the following command from the Master:

```
$ sudo salt 'test*' state.apply
```

This command applies all relevant states—as defined in our `top.sls` file—to systems whose hostnames begin with `test`. The output looks like so: (next pages)

```
$ sudo salt 'test*' state.apply
test0.iad0.tribdev.com:
```

```
-----
ID: /etc/motd
Function: file.managed
Result: True
Comment: File /etc/motd is in the correct state
Started: 23:01:06.397619
Duration: 581.967 ms
Changes:
```

```
-----
ID: redis_pkgs
Function: pkg.installed
Result: True
Comment: The following packages were installed/updated: redis
Started: 23:01:07.450629
Duration: 1107.676 ms
Changes:
```

```
-----
redis:
```

```
-----
new:
3.2.6
old:
```

```
-----
ID: /usr/local/etc/redis.conf
Function: file.managed
Result: True
Comment: File /usr/local/etc/redis.conf updated
Started: 23:01:08.560307
Duration: 535.906 ms
Changes:
```

```
-----
diff:
---
+++
@@ -1,1052 +1,82 @@
-# Redis configuration file example.
```

```
< omit multi-page diff of redis.conf>
```

```
-----
ID: /var/db/redis/
Function: file.directory
Result: True
Comment: Directory /var/db/redis is in the correct state
Started: 23:01:09.096296
Duration: 0.803 ms
Changes:
```

```
-----
ID: /etc/rc.conf.d/redis
Function: file.managed
Result: True
Comment: File /etc/rc.conf.d/redis updated
Started: 23:01:09.097172
Duration: 521.206 ms
Changes:
```

CONTINUES NEXT PAGE

```
diff:
New file
mode:
0644
```

```
Summary for test0.iad0.tribdev.com
```

```
-----
Succeeded: 5 (changed=3)
Failed:    0
-----
```

```
Total states run:      5
Total run time:    2.748 s
```

```
-----
Summary
-----
```

```
# of minions targeted: 1
# of minions returned: 1
# of minions that did not return: 0
# of minions with errors: 0
-----
```

Success! We installed the Redis binary, deployed our custom configuration files for redis and also ensured `/etc/motd` was deployed and up-to-date. I removed a very long diff that SaltStack reported when the default `redis.conf` was overwritten. Let's take a look at that configuration though, and ensure that our template and Grain data was applied correctly:

```
$ head -n 10 /usr/local/etc/redis.conf
protected-mode no
port 6379
tcp-backlog 511
bind 10.3.16.51
timeout 0

tcp-keepalive 300
$ ifconfig xn0 | grep inet
    inet 10.3.16.51 netmask 0xffffffff broadcast 10.3.16.255
$
```

And there we have it! The 0th value of the `ipv4` grain has been substituted in our configuration file and it matches the IPv4 address that is associated with `xn0` on this host.

## Conclusion

This article has barely scratched the surface of SaltStack's capabilities as a configuration management engine. Despite that, hopefully it has given a good overview of its capabilities and demonstrated the flexibility of this tool. In practice, there are several key concepts and components that I

have not covered here that the user will want to explore. We'll touch on them briefly, giving you a chance to read up on the SaltStack documentation yourself.

The first concept I completely ignored was `nodegroups`, which is a method for grouping nodes of similar function together. For example, if I have 10 Minions who act as Apache servers, I

could create a `nodegroup` in my Master configuration named `'webservers'`. I would then be able to reference this `nodegroup` in state files, thus not constraining me to hostname regular expressions as my previous example demonstrated. Here is an example of referencing a `nodegroup` in my `top.sls`.

```
dev:
  web-servers
  - match: nodegroup
  - dev_base
```

The above example will apply the `"dev_base"` salt state to all Minions I've put in my `"webservers"` `nodegroup`.

The second feature I bypassed in my demo above was Salt Pillars. They are very similar to Salt Grains, in that they are a key/value registry that



can be queried via statefiles or templates. What sets Pillars apart from Grains is the fact that they are used for user-defined variables. One common use-case would be to store DB credentials as a Pillar key/value pair, which is then referenced by a configuration template on a Minion. Furthermore, SaltStack has a GPG renderer that takes GPG-encrypted input and renders it in un-encrypted form in the Pillar for use by Minions. This method allows administrators to store credentials in a SCCS, yet ensures the encrypted payload itself is protected. •

**PETER WRIGHT** is a systems architect currently working at Tronc Inc. helping build scalable and secure systems based on FreeBSD, SaltStack and AWS for the publishing industry. A longtime member of NYCBUG, despite living in Santa Monica, he is always keen to introduce people to FreeBSD. If he's not hacking Unix, you'll probably find him at his favorite beach surfing with his son.

## Links

Getting Started with SaltStack (official tutorial):  
<https://docs.saltstack.com/en/getstarted/>

SaltStack Documentation:  
<https://docs.saltstack.com/en/latest/contents.html>

SaltStack Nodegroups:  
<https://docs.saltstack.com/en/latest/topics/targeting/nodegroups.html>

SaltStack GPG Renderer:  
<https://docs.saltstack.com/en/latest/ref/renderers/all/salt.renderers.gpg.html>



# Write For Us!



## FreeBSD<sup>TM</sup> JOURNAL



Contact Jim Maurer ([jmaurer@freebsdjournal.com](mailto:jmaurer@freebsdjournal.com))  
with your article ideas.

# svn UPDATE

by Steven Kreuzer

The goal of having a reproducible build system is fairly simple: when you compile some source code, the binary that is produced should be exactly identical to any binary compiled in the past or the future. One of the biggest advantages of reproducible builds is that you can be confident any code you run in production is exactly what you expect it to be. But it also offers numerous other benefits, such as allowing for much greater cache sharing that results in quicker build times and even solves complex issues of developers having to maintain cryptographic keys to sign code or trust the machine on which a binary was built. Developers started work to add support for creating reproducible builds back in 2013, and in the past couple months, we've seen several commits that bring this work even closer to completion. For this column, I've decided to highlight several changes, both big and small, as a way to showcase the work that goes into such a complex development project. While these types of projects tend to be invisible to the end user, it's these types of tasks that help build such a robust and verifiable system.

Add WITH\_REPRODUCIBLE\_BUILD src.conf(5) knob to disable kernel metadata.  
<https://svnweb.freebsd.org/changeset/base/310128>

Build loaders reproducibly when WITH\_REPRODUCIBLE\_BUILD.  
<https://svnweb.freebsd.org/changeset/base/310268>

Remove '-vd' option to make iasl(8) reproducible.  
<https://svnweb.freebsd.org/changeset/base/311529>

Replace non-reproducible \_\_DATE\_\_/\_\_TIME\_\_ with hardcoded string in vchi driver.  
<https://svnweb.freebsd.org/changeset/base/310560>

Avoid use of \_\_DATE\_\_ to make build reproducible in mlx driver.  
<https://svnweb.freebsd.org/changeset/base/310425>

Remove srand() to ensure deterministic output in bhnd(4).  
<https://svnweb.freebsd.org/changeset/base/310371>

Add -R option to include metadata only for unmodified src tree in newvers.sh.  
<https://svnweb.freebsd.org/changeset/base/310273>

Add option to eliminate kernel build metadata in newvers.sh.  
<https://svnweb.freebsd.org/changeset/base/310112>

Make output reproducible in makewhatis.  
<https://svnweb.freebsd.org/changeset/base/307003>

Use changelog date rather than file modification date in man pages.  
<https://svnweb.freebsd.org/changeset/base/306740>

Set UEFI boot loader PE/COFF timestamps to known value for reproducible builds.  
<https://svnweb.freebsd.org/changeset/base/305160>

However, reproducible builds are not the only exciting change to hit the src tree this month. Version 3.9.1 of LLVM was also imported, and the entire FreeBSD userland world + kernel successfully linked with LLVM's LLD (with one FreeBSD patch to be committed for an 18-year-old bug).

Upgrade our copies of clang, llvm, lld, lldb, compiler-rt and libc++ to 3.9.1 release.  
<https://svnweb.freebsd.org/changeset/base/310194>

btxldr: process all PT\_LOAD segments, not just the first two.  
<https://svnweb.freebsd.org/changeset/base/310702>

Finally, Amazon announced that they have rolled out IPv6 support in EC2 to 15 regions, and now future FreeBSD releases will support IPv6 by default on EC2. Support for IPv6 is possible on existing EC2 instances, but it will require you to run a few simple commands.

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, daughter, and dog.

COME TO OTTAWA, CANADA FOR THE 14TH ANNUAL BSDCAN!



# BSDCan 2017

## The BSD Conference

The technical conference for people working on and with 4.4BSD based operating systems and related projects. The organizers have found a fantastic formula that appeals to a wide range of people from extreme novices to advanced developers.

### WHEN

**Tutorials; June 7 & 8 (Wed/Thurs)**  
**Conference; June 9 & 10 (Fri/Sat)**

### WHERE

**University of Ottawa, in the DMS (Desmarais) building.**

**[BSDCan.org/2017](http://BSDCan.org/2017)**

# new faces

## of FreeBSD

BY DRU LAVIGNE



Johannes Dieterich



Mahdi Mokhtari

This column aims to shine a spotlight on contributors who recently received their commit bit and to introduce them to the FreeBSD community. This month, the spotlight is on **Johannes Dieterich**, who became a ports committer in January, and **Mahdi Mokhtari**, who became a ports committer in February.

Tell us a bit about yourself, your background, and your interests.

**Johannes:** I am a trained computational chemist. Our field develops simulation codes for high-performance computing (HPC) installations to study chemical processes. The relevant physics of these processes dictates what level of theory is appropriate—anything from Newtonian mechanics to various ranks of quantum mechanical approaches. One of my personal research topics is the development of global optimization techniques for chemical optimization problems.

Currently, I am a researcher at Princeton University. Method development remains my passion, and efficient implementations my obsession. When I am not coding or writing things, you will find me in a muscle car and/or exploring the American outdoors.

**Mahdi:** I'm Mahdi Mokhtari, aka MMokhi among my friends. I started programming with computers in a robotics course in high school where I learned C, and about half a year after that I started C++ in parallel. Then I was attracted to GNU/Linux, and right after that, I had an interest in developing with FreeBSD.

**How did you first learn about FreeBSD and what about FreeBSD interested you?**

**Johannes:** I worked as a system administrator (SuSE and Debian) in college to make money. I was somewhat disappointed with these sys-

tems, wondered if there was anything different and more suitable for my personal computers and came across FreeBSD. I ended up ordering a FreeBSD 6.1 CD from a retailer, a copy of Greg Lehey's *The Complete FreeBSD* and Dru Lavigne's *BSD Hacks*, which are two excellent works. After a learning curve involving a lot of recompilation of ports and broken X11, FreeBSD became my workhorse. FreeBSD was and is transparent to work with, very developer friendly, and came with a huge ports collection. The Handbook is an excellent resource for a beginner, and I found the community to be very friendly and professional to interact with.

**Mahdi:** Unlike many people, and because I'm probably the youngest person in the room (even younger than Beastie), I am not an old-fashioned Unix person who experienced the 1990s. I heard about BSD from a colleague while I was working as a part-time R&D developer. That was in the context of a comparison of FreeBSD with Linux, and I became more curious about it. I started searching and asking more people, and I found the helpful community of FreeBSD—at first on IRC.

I started asking many, many questions, from simple usage hints to functionalities of specific macros of mbufs on the kernel pf module. People on IRC answered me, and I felt I needed to replace my laptop operating system.



---

## How did you end up becoming a committer?

**Johannes:** Coming from HPC, one of the most important topics for a few years had been GPU acceleration. FreeBSD still is lagging severely behind the competition in this regard. This problem extends beyond just FreeBSD as an HPC system—highly improbable—to using it as an HPC developer platform, or even just to enable mainstream accelerated applications. I got involved with some of the efforts trying to improve this situation, especially for AMD GPUs, thinking it would be a quick thing to fix some ports. Of course, it ended up being much more work and is still ongoing. However, I have had the privilege of getting to know and work with a bunch of brilliant people and have learned a lot. This got me a commit bit and the motivation to push forward even more. My goal is to do my share in ensuring FreeBSD remains the best option as a (HPC) developer system just as much as it was in 2006 when I installed 6.1-RELEASE.

**Mahdi:** I'll begin with how I started as a contributor. After I was tied to this operating system and the community around it, I wanted to be part of it. As well as using it, I wanted to develop it! So I asked other committers how to do that. At that time, it seemed like a cliché, but now I totally understand the answer I got: "work on what you like and what needs working on, and after some time you'll see people contacting you to join."

After that, I was playing with concepts I learned, trying to port Linux applications to FreeBSD, customizing the kernel, and writing modules for it. One day, I was working on porting MySQL 5.7 for FreeBSD. I knew enough C/C++/CMake/MySQL to read and edit code as needed, and I was working on this port just out of curiosity. I also saw demands for this port on the ports mailing list, so I opened a PR ([https://bugs.freebsd.org/bugzilla/show\\_bug.cgi?id=204607](https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=204607)). With the help of bnrnd@, I successfully finished my first port and accepted maintainership of it. This was my starting point.

After this, I learned more and focused on what I loved, and tried to help the Project on things I was a so-called expert in, or on things I liked to learn about! Yes, it is a great thing that this



---

Pick a field in FreeBSD that you feel strongly about, think suboptimal, and improve it. There is a lot to be done and precious little time. Do not be shy to ask for help.” —JOHANNES

---

utopia exists!

About two years later, I was still enjoying the cycle of helping --> learning --> helping. I was contacted by feld@, who asked me if I would like to learn other things and commit my changes directly. I enthusiastically answered "Yay."

---

## How has your experience been since joining the FreeBSD Project? Do you have any advice for readers who may be interested in also becoming a FreeBSD committer?

**Johannes:** My experience has been very good indeed. I found developers behave professionally behind closed doors, in private, and in person. All the positive reactions I was greeted with were overwhelming.

Readers should take my advice, as a very junior member, with a grain of salt: I would say pick a field in FreeBSD that you feel strongly about, think suboptimal, and improve it. There is a lot to be done and precious little time. Do not be shy to ask for help.

**Mahdi:** My experience since the first question I asked has been great. People are ready to help you on PRs most of the time. Remember, they have jobs, which sometimes means a little waiting is needed.

If you like to learn, FreeBSD is one of the best places. For readers who are interested, continue working on what you love and it will benefit you and others. It just takes some time, patience, and interest. Be sure people appreciate your work, and one day you'll be punished with that commit bit.

---

**DRU LAVIGNE** is a doc committer for the FreeBSD Project and Chair of the BSD Certification Group.

# conference **REPORT**

by Benedict Reuschling

## FOSDEM 2017

FOSDEM (Free and Open Source Software Developers' European Meeting) is a noncommercial, volunteer-organized European event, centered on free and open-source software development. It is held annually, usually during the first weekend of February, at the Université Libre de Bruxelles Solbosch campus in the southeast of Brussels, Belgium.



Allan Jude and  
Kristof Provost at  
the FreeBSD table

Last year's one-day FreeBSD devsummit—held during the first day of the conference—was very well received. This year, we decided to add an extra day for that—the Friday right before FOSDEM—so there was no overlap with any talks participants might want to attend (or give), and it gave everyone an opportunity to catch up with new development efforts. The FreeBSD Foundation generously sponsored the lunch and hotel conference room.

On Friday, a total of 11 developers met in that conference room, and after a few announcements, we went right into listing topics of interest to those present. The following is what we came up with:

- libifconfig (librarification of ifconfig)
- Wayland (for which a port is now available in FreeBSD)
- GSOC participation and topics (brainstorming ideas)
- FreeBSD support for the Olimex Laptop; see this blog post for details: <https://olimex.wordpress.com/2017/02/01/teres-i-do-it-yourself-open-source-hardware-and-software-hackers-friendly-laptop-is-complete/>
- pkg flavors and subpackages (further advances to pkg(8))
- OpenSSL in base private (ports will always have to use OpenSSL from ports), or replace OpenSSL with embedTLS
- JuniorJobs and Ideas pages (updating and refreshing)

Here are some of the results that were accomplished either during the devsummit itself or over the following days:

- Removal of translation project download pages that had outdated releases. In some cases, this resulted in ridiculous releases like FreeBSD 11 for alpha, ia64, and pc98. These are not supported architectures anymore, but translation work has not caught up with reality yet, and so we decided it was better to remove these pages. In one case, translators were quick to respond and fix these issues, and pages in their languages were activated again.
- Removal of the mirror selection drop-down menu on [www.freebsd.org](http://www.freebsd.org). In the past, this was used to select a mirror that was closer to one's own location to decrease page-load times. Nowadays, the FreeBSD server infrastructure automatically selects the closest server, so there is no need for manually selecting these mirrors (which were outdated more often than not, anyway).
- Removal of bdes(1): <https://svnweb.freebsd.org/changeset/base/313329>

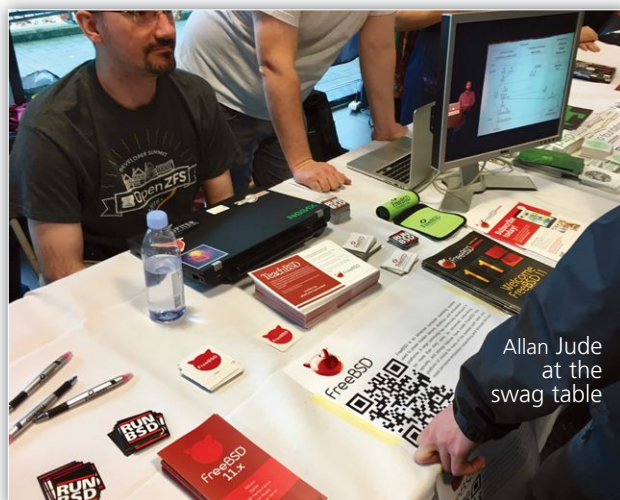
For me, this devsummit provided an excellent opportunity to do some face-to-face mentoring with Sevan Janiyan. We discussed how to do merges from HEAD to stable/11 for a man page, and ran into some issues where the merge was not done correctly and produced unexpected results. Sevan followed up on this the next day with another committer who was more familiar with Subversion internals. Apparently, we triggered a rare bug that is now being investigated further by constructing a test scenario. Luckily, a temporary local workaround was found and Sevan could continue his work.

Breaks in the morning and afternoon allowed

devsummit attendees to continue discussions over coffee and local pastries. After caffeine reserves had been replenished, we returned to our individual work or proceeded to refine GSoC ideas.

The devsummit dinner that evening allowed people who planned to attend FOSDEM the next day to join us and talk about a wide range of BSD-related topics over food and Belgian beer. A special guest was Groff, the BSD Goat, accompanied by Peter Hessler, who told us about where the two had spent the last few months (you can follow it on twitter: <https://twitter.com/GroffTheBSDGoat>).

On Saturday morning, packed with flyers and promotional material for our FreeBSD table, Allan Jude, Marie Helene Kvello-Aune, and I went to the ULB University in Brussels for the first day of FOSDEM. When we arrived, a lot of other tables had already been set up, but with fixed table designations, this was not a problem. Each table had a poster in front with a description of the project and a big QR-code that people could scan to learn more. The FreeBSD Foundation sent us marketing material like flyers, stickers, pens, and other goodies to hand out. Kristof Provost brought the rest, which had been stuck in Belgian customs, but which he managed to rescue. While setting up the



table, we were greeted and offered help by our friends from the Xen Project, some of whom we had met at dinner the night before.

Our table neighbors were from the illumos project and it wasn't long until we got into exchanges on topics of mutual interest like OpenZFS and DTrace. It was nice to see the FreeBSD bootloader running on their laptops—a result of a past collaboration between the two projects. Over the course of the weekend, we

found ourselves telling people about the productive relationship the two projects enjoy and the technologies and philosophies we share.

The halls began to quickly fill, and before we knew it, we had visitors asking all kinds of questions about FreeBSD. I had to leave the table to Allan and Kristof so I could proctor the BSDA exam in the morning. When I got back around lunchtime, FOSDEM was in full swing and a huge crowd had gathered where the open-source projects tables were located. It was a good thing we had a lot of FreeBSD people available to help! Overall, visitors were interested and open-minded about what our project had to offer.

There were attendees telling us how well FreeBSD was running on their machines and that they rarely have any maintenance issues. We told them about the newest features in FreeBSD, and most were intrigued. Some had never heard about FreeBSD, but had heard of FreeNAS or were even running it. ZFS was a discussion topic where we could show people that FreeBSD is a viable storage solution for home and enterprise use.

We had a lot of questions from people who wanted to switch from Linux distributions and asked how difficult it would be. Hardware support with drivers was a major issue in the past, but we assured them that the project supports a lot of hardware and that things are better now than just a few years back. Others were interested in FreeBSD as an embedded platform, and we told them what vendors like Netflix or iXsystems are capable of when they choose the BSD license for their products.

Other BSD activities included some interesting talks, e.g., Brooks Davis on Everything You Always Wanted to Know About "Hello, World", Ed Schouten on CloudABI, and Arun Thomas on RISC-V.

On the afternoon of the first day, there was a BSD devroom at FOSDEM. This is a separate room where projects give presentations about a specific topic. Rodrigo Osorio organized the devroom, and a video crew provided by FOSDEM recorded the talks. They are available at <https://fosdem.org/2017/schedule/track/bsd/> for people who could not make it to the event. The presentations were well attended and provided another good opportunity to show people what BSD systems can do and the new features that have been developed since last year.

Meanwhile, the FreeBSD table remained well visited and we intentionally had to hold back a couple of popular giveaways lest we run out

## conference report • FOSDEM 2017

before the end of day. We wanted to give Sunday's attendees the chance to get them as well. We had more than enough flyers, though, and handed out as many as we could. On Saturday evening, I went with a group of FreeBSD people to a restaurant outside of town that specializes in cooking with beer, and definitely, this was a culinary experience that will have to be repeated next year.

Sunday felt a little quieter than Saturday, but still was well enough attended to keep us busy handing out flyers to people and answering questions. Sometimes people recognized Allan Jude at our table and could not believe that the former TechSnap cohost and BSDNow.tv host was in Europe. This proved to be a great door opener, as people were interested in talking with him and that gave us an opportunity to show them FreeBSD.

After dinner on the last day of the event, Allan, Sevan, and I decided to do a little bit of hacking as a small group. We sat in the hotel and worked on some PRs and leftover items (also known as the impromptu hacker lounge).

Thanks to the organizers, staff, and volunteers at FOSDEM for making it such a great event. Thanks to Rodrigo Osorio for organizing and managing the BSD devroom. Thanks to everyone who helped at the FreeBSD table, to all those who attended talks, spoke with us, and offered feedback. Special thanks to Kristof Provost for organizing the FreeBSD devsummit and to the FreeBSD Foundation for sponsoring it. ●

---

**BENEDICT REUSCHLING** joined the FreeBSD Project in 2009. After receiving his full documentation commit bit in 2010, he actively began mentoring other people to become FreeBSD committers. He is a proctor for the BSD Certification Group and joined the FreeBSD Foundation in 2015, where he is currently serving as vice president. Benedict has a Master of Science degree in Computer Science and is teaching a UNIX for software developers class at the University of Applied Sciences, Darmstadt, Germany.

# Thank you!

The FreeBSD Foundation would like to acknowledge the following companies for their continued support of the Project. Because of generous donations such as these we are able to continue moving the Project forward.



Are you a fan of FreeBSD? Help us give back to the Project and donate today! [freebsdfoundation.org/donate/](http://freebsdfoundation.org/donate/)

Please check out the full list of generous community investors at [freebsdfoundation.org/donate/sponsors](http://freebsdfoundation.org/donate/sponsors)

Uranium

The Koum Family  
Anonymous

Iridium



Platinum



Gold



facebook NETFLIX

Silver





# THE INTERNET NEEDS YOU

**GET CERTIFIED AND GET IN THERE!**  
**Go to the next level with**



Getting the most out of  
BSD operating systems requires a  
serious level of knowledge  
and expertise ● ● ● ● ● ● ● ●

## **SHOW YOUR STUFF!**

Your commitment and  
dedication to achieving the  
**BSD ASSOCIATE CERTIFICATION**  
can bring you to the  
attention of companies  
that need your skills.

## **NEED AN EDGE?**

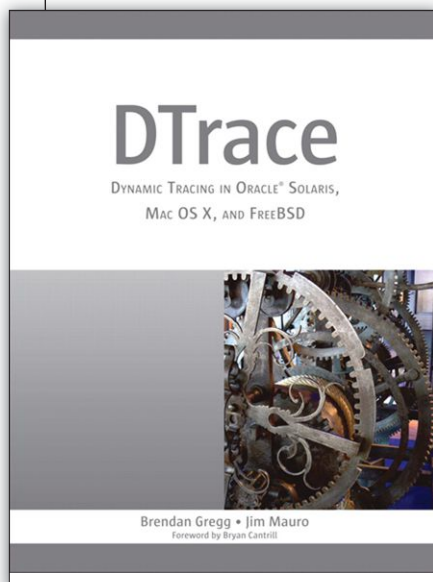
- **BSD Certification can  
make all the difference.**  
Today's Internet is complex.  
Companies need individuals with  
proven skills to work on some of  
the most advanced systems on  
the Net. With BSD Certification  
**YOU'LL HAVE  
WHAT IT TAKES!**

# **BSDCERTIFICATION.ORG**

Providing psychometrically valid, globally affordable exams in BSD Systems Administration

# BOOKreview

by Joseph Kong



## ***DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD***

by Brendan Gregg and Jim Mauro

Publisher .....	Prentice Hall (2011)
Print List Price .....	\$59.99
Digital List Price .....	\$47.99
ISBN-10 .....	0132091518
ISBN-13 .....	9780132091510
Pages .....	1,152

My coworkers love DTrace. They constantly extol its ability to root cause bugs. As such, I purchased a copy of *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, by Brendan Gregg and Jim Mauro, to bring myself up to speed (okay, I was a little jealous of my coworkers' DTrace skills). For the most part I liked *DTrace*, but there were several chapters that were a chore to slog through (more on that later).

The book is organized into three parts. The first part is comprised of two chapters and provides an overview of DTrace. Chapter 1, Introduction to DTrace, describes what exactly DTrace is and how it works. (DTrace is a dynamic tracing framework that lets you observe your system's behavior.) Chapter 2, D Language, describes how to write DTrace scripts and one-liners, detailing the individual components that make up a DTrace script. After reading this part and going through some examples online, I was equipped with enough knowledge to start using DTrace successfully at work.

The second part is comprised of eight chapters, each describing a different area that DTrace can observe. They are:

- Chapter 3, System View
- Chapter 4, Disk I/O
- Chapter 5, File Systems
- Chapter 6, Network Lower-Level Protocols
- Chapter 7, Application-Level Protocols

- Chapter 8, Languages
- Chapter 9, Applications
- Chapter 10, Databases

These chapters are jam-packed with examples and case studies, and I personally love the learn-by-example approach. They make obvious the power of DTrace. But, because most of the content wasn't readily applicable to me and because each chapter is incredibly long, it felt like a grind to read them (I now understand why my editors at No Starch always told me to avoid writing super-long chapters). In my opinion, these chapters work better as reference material.

The third and final part is comprised of four chapters, each describing a topic that didn't fit into the previous two parts. They are:

- Chapter 11, Security, which describes how to apply DTrace for real-time forensics, custom auditing, policy enforcement, and security debugging
- Chapter 12, Kernel, which details how to use DTrace to peer into your operating system's kernel; there's some overlap between this chapter and those in the second part
- Chapter 13, Tools, which describes some tools built using DTrace
- Chapter 14, Tips and Tricks, which provides some insights on how to use DTrace effectively based on the authors' experiences

Again, these chapters are jam-packed with examples. And again, I love this approach. Unlike the second part, however, these chapters are of

reasonable length. As such, the material is digestible.

Overall, *DTrace* is a good book. The first part alone is enough to bring you up to speed, and the third part was a fun read. However, this may not be a book you read cover to cover. The bulk of the book (i.e., the second part) is better served as reference material. (*DTrace* also contains seven appendices, which are obviously designed to be reference material.) ●

**JOSEPH KONG** is the author of the critically acclaimed *Designing BSD Rootkits* and *FreeBSD Device Drivers*. He is currently a senior software engineer for Dell EMC's Isilon division. For more information about Joseph Kong visit [www.thestackframe.org](http://www.thestackframe.org) or follow him on Twitter @JosephJKong.

You already know that FreeBSD is an internationally recognized leader in providing a high performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the project.

**Support  
FreeBSD**



<https://www.freebsdoundation.org/donate>

**Making a Donation is Quick & Easy.  
It Helps to Support:**

- Project Development • FreeBSD Advocacy
- Growth of the *Journal* • And Much More!

**Donate  
Today  
to the  
Foundation!**



## ZFS experts make their servers **ZING!**

Now you can too. Get a copy of.....

**Choose ebook, print or combo. You'll learn:**

- Use boot environments to make the riskiest sysadmin tasks boring.
- Delegate filesystem privileges to users.
- Containerize ZFS datasets with jails.
- Quickly and efficiently replicate data between machines
- Split layers off of mirrors.
- Optimize ZFS block storage.
- Handle large storage arrays.
- Select caching strategies to improve performance.
- Manage next-generation storage hardware.
- Identify and remove bottlenecks.
- Build screaming fast database storage.
- Dive deep into pools, metaslabs, and more!

**Link to:**

**<http://zfsbook.com>**

WHETHER YOU MANAGE A SINGLE SMALL SERVER OR INTERNATIONAL DATACENTERS, SIMPLIFY YOUR STORAGE WITH **FREEBSD MASTERY: ADVANCED ZFS**. GET IT TODAY!





THROUGH JUNE 2017

BY DRU LAVIGNE

# Events Calendar

The following BSD-related conferences will take place in the second quarter of 2017.



## LinuxFest NorthWest • May 6 & 7 • Bellingham, WA

<https://www.linuxfestnorthwest.org> • This is the 18th year for this annual, community-based conference. There will be a FreeBSD booth in the expo area, and Arun Thomas will give a presentation on RISC-V architecture. This event is free to attend.

## OSCON • May 8–11 • Austin, TX

<https://conferences.oreilly.com/oscon/oscon-tx> • Join us at the FreeBSD booth at the OSCON Expo Hall. Save 20% on your registration fee by using discount code USRG. Registration is required to attend. See you there!



## Kansas LinuxFest • May 13 & 14 • Wichita, KS

<http://kansaslinuxfest.org/> • This is the 3rd year for this annual, community-based conference. There will be a FreeBSD booth in the expo area and several BSD-related presentations. This event is free to attend.



## BSDCan • June 7–10 • Ottawa, ON

<http://www.bsdcn.org> • The 14th annual BSDCan will take place in Ottawa, Canada. This popular conference appeals to a wide range of people from extreme novices to advanced developers of BSD operating systems. The conference includes a Developer Summit, Vendor Summit, Doc Sprints, tutorials, and presentations. The BSDA certification exam will also be available. Registration is required for this event.

## SouthEast LinuxFest • June 9–11 • Charlotte, NC

<http://www.southeastlinuxfest.org/> • The 9th annual SouthEast LinuxFest is a community event for anyone who wants to learn more about open-source software. There will be several FreeBSD-related presentations and a FreeBSD booth in the expo area. There is a nominal registration fee for this conference.

